Sandrine Blazy
Christine Paulin-Mohring
David Pichardie (Eds.)

LNCS 7998

Interactive Theorem Proving

4th International Conference, ITP 2013
Rennes, France, July 2013
Proceedings

Springer

Lecture Notes in Computer Science 7998

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

Sandrine Blazy Christine Paulin-Mohring
David Pichardie (Eds.)

Interactive Theorem Proving

4th International Conference, ITP 2013
Rennes, France, July 22-26, 2013
Proceedings

Springer

Volume Editors

Sandrine Blazy
Université Rennes 1, IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
E-mail: sandrine.blazy@irisa.fr

Christine Paulin-Mohring
Université Paris-Sud, LRI
Bat 650, Univ. Paris Sud
91405 Orsay Cedex, France
E-mail: christine.paulin@lri.fr

David Pichardie
Inria Rennes-Bretagne Atlantique
Campus de Beaulieu
35042 Rennes Cedex, France
E-mail: david.pichardie@inria.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39633-5 e-ISBN 978-3-642-39634-2
DOI 10.1007/978-3-642-39634-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013942669

CR Subject Classification (1998): I.2.3, F.4.1, F.4.3, I.2.2, I.2.4, F.3, D.2.4
F.1.1, K.6.5

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

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

Printed on acid-free paper

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

Preface

This volume contains the papers presented at ITP 2013, the 4th International Conference on Interactive Theorem Proving. The conference was held during July 23–26 in Rennes, France.

ITP brings together researchers working in interactive theorem proving and related areas, ranging from theoretical foundations to implementation aspects and applications in program verification, security, and formalization of mathematics. ITP 2013 was the fourth annual conference in this series. The first meeting was held during July 11–14, 2010, in Edinburgh, UK, as part of the Federated Logic Conference (FLoC). The second meeting took place during August 22–25, 2011, in Berg en Dal, The Netherlands. The third meeting was held during August 13–15 in Princeton, New Jersey, USA. ITP evolved from the previous TPHOLs series (Theorem Proving in Higher-Order Logics), which took place every year from 1988 to 2009.

There were 66 submissions to ITP 2013, each of which was reviewed by at least three Program Committee members. Out of the 66 submissions, 53 were regular papers and 13 were rough diamonds. This year, the call for papers requested submissions to be accompanied by verifiable evidence of a suitable implementation. In accordance with this, almost all submissions came with the source files of a corresponding formalization, which influenced the acceptance decisions. The Program Committee accepted 33 papers, which include 26 regular papers and seven rough diamonds, all of which appear in this volume. We were pleased to be able to assemble a strong program covering topics such as program verification, security, formalization of mathematics, and theorem prover development. The Program Committee also invited three leading researchers to present invited talks: Dominique Bolignano (Prove & Run, France), Rustan Leino (Microsoft Research, USA), and Carsten Schürmann (IT University of Copenhagen, Denmark). In addition, the Program Committee invited Assia Mahboubi and Enrico Tassi (Inria, France) to give a tutorial on the Mathematical Components library and Panagiotis Manolios (Northeastern University, USA) to give a tutorial on counterexample generation in interactive theorem provers. We thank all these speakers for also contributing articles to these proceedings.

ITP 2013 also featured two associated workshops held the day before the conference: the AI4FM 2013 workshop and the Coq Workshop 2013. The work of the Program Committee and the editorial process were facilitated by the EasyChair conference management system. We are grateful to Springer for publishing these proceedings, as they have done for all ITP and TPHOLs meetings since 1993.

Many people contributed to the success of ITP 2013. The Program Committee worked hard at reviewing papers, holding extensive discussions during the on-line Program Committee meeting, and making final selections of accepted papers and invited speakers. Thanks are also due to the additional reviewers

enlisted by Program Committee members. Finally, we would like to thank our sponsors: Inria, the University of Rennes 1, SISCom Bretagne, Rennes Metropole and Région Bretagne.

May 2013

Sandrine Blazy
Christine Paulin-Mohring
David Pichardie

Organization

Program Committee

Wolfgang Ahrendt	Chalmers University, Sweden
Jeremy Avigad	Carnegie Mellon University, USA
Nick Benton	Microsoft Research, UK
Lennart Beringer	Princeton University, USA
Sandrine Blazy	Université Rennes 1, France
Adam Chlipala	MIT, USA
Thierry Coquand	Chalmers University, Sweden
Amy Felty	University of Ottawa, Canada
Ruben Gamboa	University of Wyoming, USA
Herman Geuvers	Radboud University Nijmegen, The Netherlands
Elsa Gunter	University of Illinois at Urbana-Champaign, USA
David Hardin	Rockwell Collins, Inc., USA
John Harrison	Intel Corporation, USA
Gerwin Klein	NICTA and UNSW, Australia
Assia Mahboubi	Inria - École polytechnique, France
Panagiotis Manolios	Northeastern University, USA
Conor Mcbride	University of Strathclyde, UK
Cesar Munoz	NASA, USA
Magnus O. Myreen	University of Cambridge, UK
Tobias Nipkow	TU München, Germany
Michael Norrish	NICTA and ANU, Australia
Sam Owre	SRI International, USA
Christine Paulin-Mohring	Université Paris-Sud 11, France
Lawrence Paulson	University of Cambridge, UK
David Pichardie	Inria Rennes, France
Brigitte Pientka	McGill University, Canada
Laurence Pierre	TIMA, France
Lee Pike	Galois, Inc., USA
Claudio Sacerdoti Coen	University of Bologna, Italy
Julien Schmaltz	Open University of the Netherlands, The Netherlands
Makoto Takeyama	AIST/COVS, Japan
René Thiemann	University of Innsbruck, Austria
Laurent Théry	Inria Sophia-Antipolis, France
Makarius Wenzel	Université Paris-Sud 11, France

Additional Reviewers

Andronick, June
Asperti, Andrea
Baelde, David
Bell, Christian J.
Bengtson, Jesper
Bertot, Yves
Blanchette, Jasmin Christian
Boldo, Sylvie
Campbell, Brian
Capretta, Venanzio
Cave, Andrew
Cohen, Cyril
Courtieu, Pierre
Dagit, Jason
Danielsson, Nils Anders
Daum, Matthias
Demange, Delphine
Diatchki, Iavor
Dénès, Maxime
Feliachi, Abderrahmane
Ferreira, Francisco
Fuhs, Carsten
Gacek, Andrew
Gammie, Peter
Greenaway, David
Gregoire, Benjamin
Huffman, Brian
Huisman, Marieke
Hur, Chung-Kil
Hölzl, Johannes
Immler, Fabian
Jackson, Paul
Joosten, Bas
Kaliszyk, Cezary
Kumar, Ramana
Lammich, Peter
Licata, Daniel R.
Lumsdaine, Peter
Matichuk, Daniel
Popescu, Andrei
Ricciotti, Wilmer
Schlesinger, Cole
Slind, Konrad
Sternagel, Christian
Stewart, Gordon
Tassi, Enrico
Verbeek, Freek
Wiedijk, Freek
Winwood, Simon
Wolff, Burkhart
Ziliani, Beta

Table of Contents

Invited Talks

Applying Formal Methods in the Large 1
Dominique Bolignano

Automating Theorem Proving with SMT 2
K. Rustan M. Leino

Certifying Voting Protocols 17
Carsten Schürmann

Invited Tutorials

Counterexample Generation Meets Interactive Theorem Proving: Current Results and Future Opportunities 18
Panagiotis Manolios

Canonical Structures for the Working Coq User 19
Assia Mahboubi and Enrico Tassi

Regular Papers

MaSh: Machine Learning for Sledgehammer 35
Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban

Scalable LCF-Style Proof Translation 51
Cezary Kaliszyk and Alexander Krauss

Lightweight Proof by Reflection Using a Posteriori Simulation of Effectful Computation 67
Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani

Automatic Data Refinement 84
Peter Lammich

Data Refinement in Isabelle/HOL 100
Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow

Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable 116
Andreas Lochbihler

Ordinals in HOL: Transfinite Arithmetic up to (and Beyond) ω_1 133
Michael Norrish and Brian Huffman

Mechanising Turing Machines and Computability Theory in Isabelle/HOL 147
Jian Xu, Xingyuan Zhang, and Christian Urban

A Machine-Checked Proof of the Odd Order Theorem 163
Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry

Kleene Algebra with Tests and Coq Tools for while Programs 180
Damien Pous

Program Analysis and Verification Based on Kleene Algebra in Isabelle/HOL 197
Alasdair Armstrong, Georg Struth, and Tjark Weber

Pragmatic Quotient Types in Coq 213
Cyril Cohen

Mechanical Verification of SAT Refutations with Extended Resolution 229
Nathan Wetzler, Marijn J.H. Heule, and Warren A. Hunt Jr.

Formalizing Bounded Increase 245
René Thiemann

Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types 261
Vincent Rahli, Mark Bickford, and Abhishek Anand

Type Classes and Filters for Mathematical Analysis in Isabelle/HOL ... 279
Johannes Hölzl, Fabian Immler, and Brian Huffman

Formal Reasoning about Classified Markov Chains in HOL 295
Liya Liu, Osman Hasan, Vincent Aravantinos, and Sofiène Tahar

Practical Probability: Applying pGCL to Lattice Scheduling 311
David Cock

Adjustable References 328
Viktor Vafeiadis

Handcrafted Inversions Made Operational on Operational Semantics 338
Jean-François Monin and Xiaomu Shi

Circular Coinduction in Coq Using Bisimulation-Up-To Techniques 354
Jörg Endrullis, Dimitri Hendriks, and Martin Bodin

Program Extraction from Nested Definitions 370
Kenji Miyamoto, Fredrik Nordvall Forsberg, and Helmut Schwichtenberg

Subformula Linking as an Interaction Method 386
Kaustuv Chaudhuri

Automatically Generated Infrastructure for De Bruijn Syntaxes 402
Emmanuel Polonowski

Shared-Memory Multiprocessing for Interactive Theorem Proving 418
Makarius Wenzel

A Parallelized Theorem Prover for a Logic with Parallel Execution 435
David L. Rager, Warren A. Hunt Jr., and Matt Kaufmann

Rough Diamonds

Communicating Formal Proofs: The Case of Flyspeck 451
Carst Tankink, Cezary Kaliszyk, Josef Urban, and Herman Geuvers

Square Root and Division Elimination in PVS 457
Pierre Neron

The Picard Algorithm for Ordinary Differential Equations in Coq 463
Evgeny Makarov and Bas Spitters

Stateless Higher-Order Logic with Quantified Types 469
Evan Austin and Perry Alexander

Implementing Hash-Consed Structures in Coq 477
Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux

Towards Certifying Network Calculus 484
Etienne Mabille, Marc Boyer, Loïc Fejoz, and Stephan Merz

Steps towards Verified Implementations of HOL Light 490
Magnus O. Myreen, Scott Owens, and Ramana Kumar

Author Index 497

Applying Formal Methods in the Large

Dominique Bolignano

Prove & Run, Paris, France

Despite intensive research done in the area of formal methods and proof techniques, these techniques remain poorly adopted and only used in marginal situations, or in niche markets. The author has been applying formal methods in industry for a few decades now. After having managed for more than ten years a company, i.e. Trusted Logic, which is applying formal methods on critical components, he has recently funded a new company, Prove & Run, whose objective is to democratize and broaden the use of formal methods. In this presentation and based on his past experience in applying formal methods and designing formal methods he describes what he believes to be the main problems, main challenges, to be overcome for achieving this. This involves in particular applying the correct formal methods to the right piece of software in the right architecture, correctly addressing the integration into the development phase, into the maintenance phase, clearly delimiting and understanding the scope of formal methods, setting up the proper organisation, finding available expertise, being consistent with the cost and time to market requirements, using the right refinements. He will explain in particular why Prove & Run will first focus on formally verifying micro-kernels, hypervisors, and Trusted Execution Environments, and which kind of properties will be proven correct.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, p. 1, 2013.

Automating Theorem Proving with SMT

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. The power and automation offered by modern satisfiability-modulo-theories (SMT) solvers is changing the landscape for mechanized formal theorem proving. For instance, the SMT-based program verifier Dafny supports a number of proof features traditionally found only in interactive proof assistants, like inductive, co-inductive, and declarative proofs. To show that proof tools rooted in SMT are growing up, this paper presents, using Dafny, a series of examples that illustrate how theorems are expressed and proved. Since the SMT solver takes care of many formal trivialities automatically, users can focus more of their time on the creative ingredients of proofs.

1 Introduction

A growing number of theorems about mathematics, logic, programming-language semantics, and computer programs are formalized and proved using mechanized proof assistants. Examples of such proof assistants are ACL2 [24], Agda [9,35], Coq [6], Guru [40], HOL Light [19], Isabelle/HOL [34], PVS [36], and Twelf [37]. The assistants vary in their level of expressivity and automation as well as in the size of their trusted computing base. Satisfiability-modulo-theories (SMT) solvers (for example, Alt-Ergo [7], CVC3 [3], OpenSMT [10], Simplify [14], and Z3 [13]) are collections of (semi-)decision procedures for certain theories. SMT solvers provide a high degree of automation and have, in the last couple of decades, undergone impressive improvements in power. Therefore, it has become increasingly common for proof assistants to use SMT solvers as subroutines, as is done for example in PVS [36] and in Isabelle/HOL's Sledgehammer tactic [8].

Although general proof assistants can be used to verify the correctness of computer programs, there are also some verification tools dedicated to verifying programs. These include Chalice [30], Dafny [26], F* [41], Frama-C [12], Hi-Lite Ada [18], KeY [4], KIV [39], Pangolin [38], Spec# [2], VCC [11], VeriFast [21], and Why3 [17]. Many of these use as their underlying reasoning engine an SMT solver, typically accessed via an intermediate verification language like Boogie [1] or Why [16]. This tool architecture facilitates automation, and it also tends to move the user's interaction with the tool from the formula level (like in general proof assistants) to the program level. This lets users express necessary proof ingredients in program-centric declarations like preconditions or loop invariants. We might therefore refer to this kind of program verifier as *auto-active*—a mix of automatic decision procedures and user interaction at the program level [28].

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 2–16, 2013.

Just as some proof assistants have incorporated special tactics to better handle program verification (*e.g.*, Ynot [33]), auto-active program verifiers are incorporating features to better support user-guided mathematical proofs (*e.g.*, VeriFast [22] and Dafny [26]). While such program verifiers do not yet achieve the full expressivity of some proof assistants and generally have a much larger trusted computing base, their automation can be remarkable. This automation matters, since it affects the amount of human time required to use the tool. Curiously, these SMT-based tools are primarily program verifiers, but that seems to have happened more by serendipity; one can easily imagine similar SMT-based tools that focus on mathematics rather than on programs.

In this paper, and in the invited talk that the paper accompanies, I argue and showcase that the future may hold room for proof assistants that are entirely based on SMT solving. I focus on the programming language and auto-active program verifier Dafny [26], which supports proof features like induction [27], co-induction [29], and declarative calculations [31]. The paper is a collection of examples that give an idea of what these proof features can do and how they are represented in the Dafny input.

Sec. 2 defines a type and a function that will be used throughout the examples. Sec. 3 states a lemma and proves it by induction. It also shows a proof calculation. Sec. 4 then turns to the infinite by considering co-inductive declarations and proofs. Sec. 5 combines the inductive and co-inductive features into a famous "filter" function, and Sec. 6 proves a theorem about filters by simultaneously applying induction and co-induction. Sec. 7 summarizes the examples and concludes.

2 A Types and a Function

As a basis for all the examples in this paper, let us define a type `Stream` whose values are infinite lists. Such a type is called a *co-inductive datatype*, but this fancy name need not cause any alarms.

```
codatatype Stream<T> = Cons(head: T, tail: Stream)
```

The stream type is parameterized by the type of its elements, `T`. The stream type has one constructor, `Cons`. The names of the parameters to the constructor (`head` and `tail`) declare destructors. For example, for any stream `s`, we have

```
s = Cons(s.head, s.tail)
```

The type of `tail` is `Stream<T>`, but here and in other signatures, the type argument `T` can be supplied automatically by Dafny, so I omit it.

Next, let us declare a function that returns a suffix of a given stream. In particular, `Tail(s, n)` returns $\texttt{s.tail}^{\texttt{n}}$. Here is its inductive definition:

```
function Tail(s: Stream, n: nat): Stream
{
  if n = 0 then s else Tail(s.tail, n-1)
}
```

Each function is checked for well-definedness. For a recursive function, this includes a check of well-foundedness among the recursive calls. Well-foundedness is checked

using a *variant function*, which conceptually is evaluated on entry to a call. If the value of the variant function is smaller for the callee than for the caller, well-foundedness follows. If no variant function is explicitly supplied, Dafny guesses one. The guess is the lexicographic ordering on the tuple of the function's arguments, omitting arguments whose types have no ordering, like co-inductive datatypes. For `Tail`, Dafny (correctly) guesses the variant function `n` and, using it, checks that `Tail`'s recursion is indeed well-founded. No further input is required from the user—the tool automatically initiates the check—and since the check succeeds, the user is not bothered by any messages from the tool (except for a small indication in the margin of the integrated development environment that for a split second shows that the verifier is active).

3 An Inductive Proof

In order to show how lemmas and proofs are set up, let us consider an alternative definition of `Tail`:

```
function Tail_Alt(s: Stream, n: nat): Stream
{
  if n = 0 then s else Tail_Alt(s, n-1).tail
}
```

and let us prove that `Tail` and `Tail_Alt` give the same result.

A lemma is expressed as a *method* in the programming language, that is, as a code procedure with a pre- and postcondition. As usual (*e.g.*, [20]), such a method says that for any values of the method parameters that satisfy the method's precondition, the method will terminate in a state satisfying the postcondition.[1] This corresponds to what is done in mathematics for a lemma: a lemma says that for any values of the lemma parameters that satisfy the lemma's antecedent, the lemma's conclusion holds. So, we express our lemma about `Tail` and `Tail_Alt` by declaring the following method:

```
ghost method Tail_Lemma(s: Stream, n: int)
  requires 0 ≤ n;
  ensures Tail(s, n) = Tail_Alt(s, n);
```

The precondition of this method (keyword **`requires`**) says that `n` is a natural number (which, alternatively, we could have indicated by declaring the type of `n` to be **`nat`**). The postcondition (keyword **`ensures`**) gives the property we want to prove. The designation of the method as a *ghost* says that we do not want the Dafny compiler to emit any executable code. In other words, a ghost method is for the verifier only; the compiler ignores it.

To get the program verifier to prove the lemma, we supply a method body and let the verifier convince itself that all code control paths terminate and establish the postcondition. The body we supply typically consists of **`if`** statements and (possibly recursive) method calls, but other statements (*e.g.*, **`while`** loops) can also be used. A recursive call

[1] In general, methods can have effects on the memory of the program. Such effects are declared with a **`modifies`** clause in the method specification. However, since there is no need for such effects here, I ignore further discussion of them.

corresponds to invoking an inductive hypothesis, because the effect on the proof is to obtain the proof goal (stated in the postcondition) for the callee's arguments, which in a well-foundedness check are verified to be "smaller" than the caller's arguments. It is common to supply assertions that act as in-place lemmas: a statement **assert** `Q` tells the verifier to check that the boolean condition `Q` holds, after which the verifier can make use of that condition. A more systematic way to direct the verifier is provided by Dafny's **calc** statement, whose verified calculations take the form of human-readable equational proofs [31].

Here is a proof of the lemma:

```
ghost method Tail_Lemma(s: Stream, n: int)
  requires 0 ≤ n;
  ensures Tail(s, n) = Tail_Alt(s, n);
{
  if n < 2 {
    // def. of Tail and Tail_Alt
  } else {
    calc {
      Tail(s, n);
    =  // def. Tail, since n ≠ 0
      Tail(s.tail, n-1);
    =  { Tail_Lemma(s.tail, n-1); }  // induction hypothesis
      Tail_Alt(s.tail, n-1);
    =  // def. Tail_Alt, since n-1 ≠ 0
      Tail_Alt(s.tail, n-2).tail;
    =  { Tail_Lemma(s.tail, n-2); }  // induction hypothesis
      Tail(s.tail, n-2).tail;
    =  // def. Tail, since n-1 ≠ 0
      Tail(s, n-1).tail;
    =  { Tail_Lemma(s, n-1); }  // induction hypothesis
      Tail_Alt(s, n-1).tail;
    =  // def. Tail_Alt, since n ≠ 0
      Tail_Alt(s, n);
    }
  }
}
```

The method body provides two code paths. For the `n < 2` branch, the verifier can prove the postcondition by unwinding the definitions of each of `Tail` and `Tail_Alt` once or twice (which the verifier is willing to do automatically). The else branch uses a **calc** statement with a number of equality-preserving steps, each of which is verified. Some steps are simple and need no further justification; the code comments give explanations for human consumption. Other steps are justified by *hints*, which are given as code blocks (in curly braces). Here, each hint makes a recursive call to `Tail_Lemma`, which in effect invokes the induction hypothesis.

In more detail, for each step in a calculation, the verifier checks that the equality entailed by the step is provable after the code in the associated hint (if any). In the

calculation above, each of the provided hints consists of a single recursive call. As usual in program verification, the verifier thus checks that the precondition of the callee is met, checks that the variant function is decreased for the call (to ensure that the recursion will terminate), and can then assume the postcondition of the callee. For example, the postcondition that holds after the first recursive call is:

```
Tail(s.tail, n-1) = Tail_Alt(s.tail, n-1)
```

which is essentially the induction hypothesis for `s,n := s.tail,n-1`. Since no variant function is supplied explicitly, Dafny guesses `n`, which it verifies to decrease. Thus, the recursion—and indeed, the induction—is well-founded.

For brevity, the equality signs between the lines in the calculation can be omitted. Or, if desired, they can be replaced by different operators, like $\Longrightarrow$, $\Longleftarrow$, or $<$.

The calculation in the example gives more detail than the Dafny verifier needs, but, as given, yields a presentation of the proof that is better suited for a human. In fact, the proof calculation is quite readable; it looks almost identical to how one would write an equational-style proof by hand. For a comparison with other styles of declarative proofs, like Isar [42], and with tactic-based proofs, see [31].

4 Co-recursion and a Co-inductive Proof

In this section, we consider how values of a co-datatype are constructed and how one states and proves properties of such values.

Values of co-inductive datatypes may be of an infinite nature. For example, a stream represents an infinite list of elements. Here is a function that defines such a value, namely the stream whose elements are the integers from `n` upward in increasing order:

```
function Up(n: int): Stream<int>
{
  Cons(n, Up(n+1))
}
```

It may look as if invocations of `Up` will never terminate, but the self-call of `Up` is identified by Dafny as being *co-recursive*, because it is positioned as an argument to a co-datatype constructor. Co-recursive calls are compiled into lazily evaluated code, so that the arguments to the constructor are not evaluated until their values are used by the executing program (if ever). Consequently, for a co-recursive call, there is no need for the verifier to enforce a decrease of a variant function.

Here is another function on streams:

```
function Prune(s: Stream): Stream
{
  Cons(s.head, Prune(s.tail.tail))
}
```

It defines a stream consisting of half of the elements of the given stream: every other element, starting with the first. Note that the self-call to `Prune` is co-recursive, so the verifier does not need to check termination.

To define a property of a co-inductive datatype, one uses a *co-predicate*. For example, the following co-predicate holds for streams that consist of even integers:

```
copredicate AllEven(s: Stream<int>)
{
  s.head % 2 = 0 ∧ AllEven(s.tail)
}
```

Co-predicates are defined by greatest fix-points, that is, as the greatest solutions of the recursive equations to which their definitions give rise. Applied to the example, this means that `AllEven(s)` evaluates to **true** as long as there is no suffix `t` of `s` such that `t.head % 2 ≠ 0`. Eager evaluation of a co-predicate may fail to terminate and lazy evaluation would not be meaningful, so co-predicates are always ghost. In other words, they are never part of executing code, but they can be used to describe and reason about executing code.

We have now seen three features from the quartet of co-inductive features in Dafny: co-datatypes define possibly infinite data structures, co-recursive function calls make it possible to define values of co-datatypes, and co-predicates define properties of co-datatypes. The fourth feature is *co-methods*, whose purpose is to enable co-inductive proofs. Let us consider an example.

We will state a theorem that for any even `n`, `Prune(Up(n))` consists only of even integers. Because we intend to prove the theorem by co-induction, we use a co-method:

```
comethod Theorem(n: int)
  requires n % 2 = 0;
  ensures AllEven(Prune(Up(n)));
{
  Theorem(n+2);
}
```

Ignoring the issue of termination, this proof can be understood in the same manner as inductive proofs: `AllEven` says something about the head of the stream `Prune(Up(n))`, which is proved automatically. It also says something about the tail of the stream, which follows from the postcondition of the call `Theorem(n+2)` and the definitions of the functions involved. To make this argument more explicit, the call could have been preceded by the following calculation (where, for brevity and variety, I have chosen to omit the optional equality signs between lines in the left margin):

```
  calc {
    Prune(Up(n)).tail;
    Prune(Up(n).tail.tail);
    { assert Up(n).tail.tail = Up(n+2); }
    Prune(Up(n+2));
  }
```

In contrast to methods, whose recursive calls are checked to terminate (by checking that they decrease the variant function), calls to co-methods are always allowed. In other words, the co-induction hypothesis can always be obtained; however, the *use* of it is restricted. Intuitively, the co-induction hypothesis can be used to discharge only

those conjuncts that show up after one unwinding of the co-predicate in the co-method's postcondition. I will give some details about this in Sec. 6.

For example, suppose the body of co-method `Theorem` were replaced by the call `Theorem(n)`. With unrestricted use of the postcondition of this call, the co-induction hypothesis obtained would trivially prove the theorem itself. However, because the co-induction hypothesis can be used only on conjuncts from an unwinding of the postcondition, the call `Theorem(n)` provides no benefit here.

5 A Filter Function

Let us now consider a more difficult function definition, namely that of a *filter* function on streams. For any stream `s`, we want the filter function to return the stream consisting of those elements of `s` that satisfy some predicate `P`. A filter function like this is used, for example, in the prime number sieve of Eratosthenes (*cf.* [5,25,15]).

Conceptually, the filter function and all related lemmas are parameterized by the predicate `P`. Lacking the higher-order features necessary to take `P` as a parameter, we represent an arbitrary predicate by declaring a (here, global) generic predicate without a defining body:[2]

```
predicate P<T>(x: T)
```

The definition of `Filter` has the following form:

```
function Filter(s: Stream): Stream
  // ... specification to be written ...
{
  if P(s.head) then
    Cons(s.head, Filter(s.tail))
  else
    Filter(s.tail)
}
```

The first branch of this definition is fine, because its call to `Filter` is co-recursive. However, the other call to `Filter` is not co-recursive, so it is subject to a termination check. This makes sense, because if the given stream has no elements that satisfy `P`, then `Filter` would never terminate in its computation to produce the next element of the resulting stream. Note, thus, how Dafny allows one function to be involved in both recursive and co-recursive calls. Next, we will consider how to deal with the termination of the recursive call.

To avoid non-termination, we must restrict `Filter`'s input to streams that contain infinitely many elements that satisfy `P`. We give the following definitions:

```
predicate HasAnother(s: Stream)
{
```

[2] Dafny allows such a body-less predicate to be placed in a *module*. Other modules can then be declared as *refinements* of this module, and each refinement module can give its own specific definition of the predicate.

```
  ∃ n • 0 ≤ n ∧ P(Tail(s, n).head)
}
copredicate AlwaysAnother(s: Stream)
{
  HasAnother(s) ∧ AlwaysAnother(s.tail)
}
```

Predicate `HasAnother(s)` says that, after some finite prefix of `s`, there is an element that satisfies `P`, and `AlwaysAnother(s)` says that `HasAnother` holds at every point in the stream. We can now restrict the input to `Filter` by adding a precondition:

```
  requires AlwaysAnother(s);
```

Even with this precondition, the verifier complains that it cannot prove termination. To remedy the situation, we supply a variant function explicitly. As the variant function, we will use the length of the non-`P` prefix of `s`, that is, the number of steps to the next element satisfying `P`. Using `StepsToNext(s)` to denote that number of steps, we add to the specification of `Filter` the following clause:

```
  decreases StepsToNext(s);
```

Given a stream that satisfies `AlwaysAnother`, function `StepsToNext` returns a natural number. It is tempting to define it with a body like

```
  if P(s.head) then 0 else 1 + StepsToNext(s.tail)
```

but to prove that this recursive call to `StepsToNext` terminates, we would need a variant function like `StepsToNext` itself. Instead, we find a number of steps that will yield some `P` element, and then we use this number as an upper bound in a linear search to the first `P` element:

```
function StepsToNext(s: Stream): nat
  requires AlwaysAnother(s);
{
  var n :| 0 ≤ n ∧ P(Tail(s, n).head);
  Steps(s, n)
}
function Steps(s: Stream, n: nat): nat
  requires P(Tail(s, n).head);
  ensures P(Tail(s, Steps(s, n)).head);
  ensures ∀ i • 0 ≤ i < Steps(s, n) ⟹ ¬P(Tail(s, i).head);
{
  if P(s.head) then 0 else 1 + Steps(s.tail, n-1)
}
```

These definitions require some explanation.

The "let such that" expression `var x :| Q; E` evaluates to `E` in which all free occurrences of `x` are bound to a value that satisfies `Q`. The expression is well-defined only if there exists a value for `x` that satisfies `Q`. In `StepsToNext`, this proviso follows from the precondition `AlwaysAnother(s)`. Note that `n` may be set to any number of steps that will reach a `P` element in `s`, not necessarily the smallest.

The specification of the auxiliary function `Steps` requires `s` to reach a `P` element in `n` steps. It ensures that the result value, which in the **`ensures`** clause is denoted by `Steps(s, n)`, is not only a number of steps that reaches a `P` element (first **`ensures`** clause) but also the smallest such number (second **`ensures`** clause).

The body of `Steps` encodes a straightforward linear search.

To prove the recursive call in the body of `Filter(s)` to be well-founded, the verifier checks that the given variant function decreases, that is,

```
StepsToNext(s.tail) < StepsToNext(s)
```

This condition rests on the fact that `StepsToNext` returns the smallest number of steps to reach a `P` element, which is spelled out by the postcondition of `Steps`. Note that `StepsToNext` does not need to declare such a postcondition, because Dafny unwinds the definition of `StepsToNext` and obtains an expression in terms of `Steps`. Since `Steps` is recursive, the needed property is not evident from any bounded number of unwindings, so the presence of the postcondition essentially facilitates an inductive argument.

Finally, rather than introducing the auxiliary function `Steps`, one could consider replacing the body of `StepsToNext` with one whose let-such-that condition is stronger:

```
var n :| 0 ≤ n ∧ P(Tail(s, n).head) ∧
         ∀ i • 0 ≤ i < n ⟹ ¬P(Tail(s, i).head);
n
```

However, Dafny is unable to prove the existence of such an `n` directly from the precondition `AlwaysAnother(s)`. The use of `Steps` is one way to set up the necessary inductive argument.

6 A Property of Filter

The interesting property to prove about `Filter(s)` is that it returns the subsequence of `s` that consists of exactly those elements that satisfy `P`. The notion of such a subsequence can be divided up into the property that `Filter` returns the right set of elements:

```
∀ x • x ∈ Filter(s) ⟺ x ∈ s ∧ P(x)
```

(where I have taken the liberty of using operator $\in$ as if stream were sets) and the property that `Filter(s)` preserves the order of elements in `s`. Let us look at one possible way to state and prove the latter.

To simplify matters, let us suppose that there is a function `Ord` from the elements of streams to the integers.

```
function Ord<T>(x: T): int
```

Using a co-predicate, we define what it means for a stream's elements to be strictly increasing:

```
copredicate Increasing(s: Stream)
{
  Ord(s.head) < Ord(s.tail.head) ∧ Increasing(s.tail)
}
```

Now we can state the order-preservation theorem that we want to prove:

```
ghost method Theorem_FilterPreservesOrdering(s: Stream)
  requires AlwaysAnother(s) ∧ Increasing(s);
  ensures Increasing(Filter(s));
```

This theorem is not the most general order-preservation theorem we can state, but it suffices for our purpose of showing an interesting proof.

To prove the theorem, we introduce an alternative definition of `Increasing`:

```
copredicate IncrFrom(s: Stream, low: int)
{
  low ≤ Ord(s.head) ∧ IncrFrom(s.tail, Ord(s.head) + 1)
}
```

The two definitions are interchangeable, as the following two lemmas show.

```
comethod Lemma_Incr0(s: Stream, low: int)
  requires IncrFrom(s, low);
  ensures Increasing(s);
{
}
comethod Lemma_Incr1(s: Stream)
  requires Increasing(s);
  ensures IncrFrom(s, Ord(s.head));
{
  Lemma_Incr1(s.tail);
}
```

The co-inductive proof of `Lemma_Incr0` is done automatically, whereas the other requires an explicit appeal to the co-induction hypothesis.

We can now write the theorem in terms of `IncrFrom`:

```
comethod Lemma_FilterPreservesIncrFrom(s: Stream, low: int)
  requires AlwaysAnother(s) ∧ IncrFrom(s, low) ∧ low ≤ Ord(s.head);
  ensures IncrFrom(Filter(s), low);
  decreases StepsToNext(s);
{
  if P(s.head) {
    Lemma_FilterPreservesIncrFrom(s.tail, Ord(s.head) + 1);
  } else {
    Lemma_FilterPreservesIncrFrom#[_k](s.tail, low);
  }
}
```

The proof of this lemma is interesting because it uses co-induction and induction together. The first branch of the **if** statement makes an appeal to the co-induction hypothesis. Dafny will actually fill it in automatically, so the proof also goes through with that call omitted. In the else branch, we cannot use the co-induction hypothesis, because, as discussed above, the co-induction hypothesis can be used only after one unwinding

of the proof goal. To make use of the lemma's postcondition for `s.tail` directly, we instead make a recursive call to the lemma. Syntactically, this is achieved by the characters "`#[_k]`". The recursive call gives rise to a proof obligation of termination, which is addressed by the explicit **`decreases`** clause.

Co-methods are used to establish the validity of co-predicates (including equality on co-datatype values, which is a built-in co-predicate). These co-inductive proof obligations are actually carried out by induction, in conjunction with a meta-theorem. For any co-predicate `Q(x)`, let the *prefix predicate* `Q#[_k](x)` denote the first `_k` unrollings of `Q`, defined inductively. For example, the prefix predicate for co-predicate `AllEven` is:[3]

```
predicate AllEven#[_k: nat](s: Stream<int>)
{
  if _k = 0 then
    true
  else
    s.head % 2 = 0 ∧ AllEven#[_k-1](s.tail)
}
```

Similarly, for each co-method `M(x)`, Dafny generates a *prefix method* `M#[_k](x)`, where each call `M(E)` in the co-method's body is turned into a call `M#[_k-1](E)` in the corresponding prefix method. In more detail, the following co-method:

```
comethod M(x: T)
  ensures Q(x);
  decreases D(x);
{
  ... M(E); ...
}
```

is turned into:

```
ghost method M#[_k: nat](x: T)
  ensures Q#[_k](x);
  decreases _k, D(x);
{
  if _k ≠ 0 {
    ... M#[_k-1](E); ...
  }
}
```

Any explicit prefix-method call in the body of `M` (like the one in the else branch of the filter lemma co-method above) is left unchanged in the corresponding prefix method. A recursive call to `M#[K]` where `K < _k` corresponds to obtaining the co-induction hypothesis (for use after `_k - K` unwindings of the co-predicate in the proof goal), whereas a call to `M#[_k]` is just an ordinary recursive call corresponding to the induction

[3] Prefix predicates are declared automatically. The made-up declaration syntax shown here is suggestive of how prefix predicates are actually invoked, with the unrolling-depth argument in square brackets, set apart from the other arguments.

hypothesis. By verifying the inductive prefix method for any _k, the postcondition of the co-method follows on account of the following meta-theorem [32,29]:

```
∀ x: T • Q(x) ⟺ ∀ _k: nat • Q#[_k](x)
```

For more details, see [29].

Finally, the proof of `Theorem_FilterPreservesOrdering` is given as follows:

```
{
  Lemma_Incr1(s);
  Lemma_FilterPreservesIncrFrom(s, Ord(s.head));
  Lemma_Incr0(Filter(s), Ord(s.head));
}
```

7 Conclusion

In this paper, I have conveyed a flavor of Dafny's proof features by showing examples of inductive and co-inductive definitions, proofs by induction and by co-induction, as well as human-readable proofs. These are features that until recently were confined to interactive proof assistants, but they can now be supported by auto-active verifiers.

To try the examples in the Dafny tool,[4] the only input given to the tool are the lines shown in this paper—no additional proof tactics need to be supplied.

The given examples showcase the high degree of automation that is possible in a tool powered by an SMT solver and designed to keep the interaction at the problem level (and not, for example, at the level of defining and using necessary prover tactics). Users are not bothered with trivial details (like the associativity of logical and arithmetic operators) and the human involvement to prove that user-defined functions are mathematically consistent is small. Even the tricky recursive call of `Filter` is solved by defining and using `StepsToNext` as a variant function, which does not require an excessive amount of human effort. When more information is needed, human-readable calculations can be used, putting proofs in a format akin to what may be done by hand.

For each function and method in the examples shown, the verifier needs to spend only a small fraction of a second. This makes performance good enough to be running the verifier continuously in the background of the integrated development environment, which is what Dafny does. Most changes of the program text yield a near-instant response, which is important when developing proofs. Note that performance is at least as important for failed proofs as for successful proofs, because failed proofs happen on the user's time (see [28] for some research directions for auto-active verification environments).

In the future, I expect a higher degree of automation to become available in proof assistants. For tools like Dafny that already provide a high degree of automation, I expect to see a richer set of features (for example, higher-order functions, drawing inspiration from Pangolin [38], Who [23], and F* [41], and user-defined theories, drawing inspiration from Coq [6] and Why3 [17]) as well as work that will seek to reduce the currently large trusted computing base for SMT-based verifiers.

[4] Dafny can be installed from `http://dafny.codeplex.com`. It can also be run directly in a web browser at `http://rise4fun.com/dafny`

Acknowledgments. I am grateful to Maria Christakis, Sophia Drossopoulou, Peter Müller, and David Pichardie for comments on an earlier draft of this paper.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Communications of the ACM 54(6), 81–91 (2011)
3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
5. 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)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
7. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: SMT 2008/BPR 2008: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, pp. 1–5. ACM (July 2008)
8. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
9. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda — a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)
10. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
11. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C — a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
15. Devillers, M., Griffioen, D., Müller, O.: Possibly infinite sequences in theorem provers: A comparative study. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 89–104. Springer, Heidelberg (1997)
16. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)

17. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013)
18. Guitton, J., Kanig, J., Moy, Y.: Why Hi-Lite Ada? In: Leino, K.R.M., Moskal, M. (eds.) BOOGIE 2011: First International Workshop on Intermediate Verification Languages (August 2011)
19. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580, 583 (1969)
21. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (August 2008)
22. Jacobs, B., Smans, J., Piessens, F.: VeriFast: Imperative programs as proofs. In: VSTTE Workshop on Tools & Experiments (August 2010)
23. Kanig, J., Filliâtre, J.-C.: Who: A verifier for effectful higher-order programs. In: ACM SIGPLAN Workshop on ML. ACM (August 2009)
24. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
25. Leclerc, F., Paulin-Mohring, C.: Programming with streams in Coq — A case study: The sieve of Eratosthenes. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 191–212. Springer, Heidelberg (1994)
26. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
27. Leino, K.R.M.: Automating induction with an SMT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 315–331. Springer, Heidelberg (2012)
28. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Ball, T., Zuck, L., Shankar, N. (eds.) UV 2010 (Usable Verification) Workshop (November 2010), http://fm.csl.sri.com/UV10/
29. Leino, K.R.M., Moskal, M.: Co-induction simply: Automatic co-inductive proofs in a program verifier. Technical Report MSR-TR-2013-49, Microsoft Research (May 2013)
30. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
31. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Fifth Working Conference on Verified Software: Theories, Tools, and Experiments, VSTTE 2013 (May 2013)
32. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
33. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Dependent types for imperative programs. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, pp. 229–240. ACM (September 2008)
34. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
35. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology (September 2007)
36. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
37. 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)

38. Régis-Gianas, Y., Pottier, F.: A Hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008)
39. Reif, W.: The KIV system: Systematic construction of verified software. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 753–757. Springer, Heidelberg (1992)
40. Stump, A., Deters, M., Petcher, A., Schiller, T., Simpson, T.: Verified programming in Guru. In: Altenkirch, T., Millstein, T. (eds.) PLPV 009 — Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification, pp. 49–58. ACM (January 2009)
41. Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 266–278. ACM (September 2011)
42. Wenzel, M.: Isabelle/Isar—A versatile environment for human-readable formal proof documents. PhD thesis, Institut für Informatik, Technische Universität München (2002)

Certifying Voting Protocols

Carsten Schürmann

IT University of Copenhagen, Copenhagen, Denmark

Since 2000, all but eleven countries in the world have held national elections. Most of these countries have been using computers in the voting process in one way or the other, for example, for checking off voters of the electoral role, for digitally recording of votes, and also for computing the social choice function. Elections are the cornerstone of representative democracies, the collective trust of the voters in the voting process legitimizes its result. Our work within the DemTech research project (www.demtech.dk) aims to maintain or even increase the level of the trust by applying modern theorem proving technology to the domain of voting protocols and schemas.

In my talk I will report on the research activities within the DemTech project and present in detail how we use proof assistants (such as Agda and Celf) to bridge the gap between abstract definitions of security, declarative descriptions of electoral law, and certifiable computations. We hope that our work will have a positive effect on voting processes around the world, make them safer, more reliable, less vulnerable to attack, and thus more trustworthy.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, p. 17, 2013.

Counterexample Generation Meets Interactive Theorem Proving: Current Results and Future Opportunities

Panagiotis Manolios

Northeastern University, USA

This tutorial will explore the integration of counterexample generation with interactive theorem proving, a capability that has long been on the wish list of users and developers of interactive theorem provers. While the generation of counterexamples is an undecidable problem, recent methods have shown that it is possible to generate counterexamples to conjectures for many interesting problems. This tutorial will review current counterexample generation technology and how it can be used to design, analyze, and reason about systems. The tutorial will include a demo using ACL2s, the ACL2 Sedan. During the tutorial, we will also discuss the pedagogical use of counterexample generation in ACL2s to help freshmen students learn about logic and program verification. Finally, we will discuss future research opportunities.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, p. 18, 2013.

Canonical Structures for the Working Coq User

Assia Mahboubi and Enrico Tassi

Inria

Abstract. This paper provides a gentle introduction to the art of programming type inference with the mechanism of Canonical Structures. Programmable type inference has been one of the key ingredients for the successful formalization of the Odd Order Theorem using the Coq proof assistant. The paper concludes comparing the language of Canonical Structures to the one of Type Classes and Unification Hints.

1 Introduction

One of the key ingredients to the concision, and intelligibility, of a mathematical text is the use of notational conventions and even sometimes the abuse thereof. These notational conventions are usually shaped by decades of practice by the specialists of a given mathematical community. If some conventions may vary according to the author's taste, most tend to stabilize into a well-established common practice. A trained reader can hence easily infer from the context of a typeset mathematical formula he is reading all the information that is not explicit in the formula but that is nonetheless necessary to the precise description of the mathematical objects at stake.

Formalizing a page of mathematics using a proof assistant requires the description of objects and proofs at a level of detail that is few orders of magnitude higher than the one at which a human reader would understand this description. This paper is about the techniques that can be used to reproduce at the formal level the ease authors of mathematics have to omit some part of the information they would need to provide, because it can be inferred. In the context of a large scale project like the formal proof of the Odd Order Theorem, which involves a large and broad panel of algebraic theories that should be both developed and combined, a faithful imitation of these practices becomes of crucial importance. Without them, the user of the proof assistant is soon overwhelmed by the long-windedness of the mathematical statements at stake.

What makes the meaning of mathematical conventions unambiguous and predictable is the fact that the human process of guessing the information that is not there can be described by an algorithm, which is usually very simple. In the Coq proof assistant, information which is not provided by the user is encoded in types, and the type inference algorithm is programmed to allow the user to omit information that can be inferred. Combining programmable type inference with user notations is a key design pattern of the libraries developed by the Mathematical Components team [18]. This combination is at the core of the overloading

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 19–34, 2013.

of notations [8, 2], but also of the hierarchy of algebraic theories [5, 4, 3], as well as of various forms of automatic proof search and quotation [7].

The language used throughout the Mathematical Components libraries to program type inference is the one of Canonical Structures. Although this feature has been implemented in Coq from early versions of the system, it is much underrated by most of the community of users. As of today Canonical Structures actually remain poorly documented, to the notable exception of the work of Gonthier, Ziliani, Nanevski and Dreyer [10], where the authors detail the many technical issues one has to master in order to program proficiently proof search algorithms using Canonical Structures. This paper aims at providing a gentler introduction to Canonical Structures for the Coq user. Therefore we do not expose new contributions here but rather try to make a clear description of the methodologies employed to build the Mathematical Components libraries, so that they hopefully become accessible in practice to a wider audience. For this purpose, all the examples we draw in what follows are written in standard Coq version 8.4, without relying on any of the Mathematical Components library nor on the SSReflect shell extension [9] these libraries have been developed with. We expect the reader to be familiar with the vernacular language of Coq. The running example can be download at `http://ssr.msr-inria.inria.fr/doc/cs4wcu.v`

2 Canonical Structures

Saïbi [14] introduced both the Canonical Structure and the coercion mechanisms for the Coq system almost 15 years ago [14]. They have been available since version 6.1 of Coq system. However, while coercions were given visibility by publications in international venues [13], the only detailed description of Canonical Structures can be found in Saïbi's Ph.D. dissertation, that is written in French. The introduction of these mechanisms was motivated by the formalization of category theory due to Saïbi and Huet [12], in order to enhance the support offered by the Coq system for the formalization of algebraic concepts. The two mechanisms were seen as dual features: coercions were used to map a rich structure into a simpler one by forgetting parts of it and canonical structures where used to enrich a structure, in order for instance to restore the content discarded by the earlier insertion of a coercion.

In their formalization [12], Saïbi and Huet use Canonical Structures to overload function symbols [14, section 4.7, page 82] and to relate an algebraic theory with its instances. This technique has been re-used in a similar way for more recent large scale formalizations like the Fundamental Theorem of Algebra [6]. In his dissertation (see the example [14, 8.11.2, page 155]), Saïbi already spots that the type inference algorithm can iterate the Canonical Structure mechanism, but he seems to fail to grasp its potential and hence does not advertise this possibility. As a consequence, the documentation of Canonical Structures in the Coq manual (see [17], chapter 2.7.17) presents this mechanism as a non iterative one, which can only be used to link abstract theories with the instances explicitly declared by the user.

In 2005, during the first year of the project toward the formalization of the Odd Order Theorem, Gonthier starts to make systematic use of Canonical Structures. In particular, he understands that the declaration of canonical instances can trigger an iterative process that lets one *program* type inference in a way that is reminiscent of Prolog. Combining this remark with the expressiveness of the type system of Coq it becomes quite natural to encode proof search into type inference.

The essence of the Canonical Structures mechanism is to extend the *unification* algorithm of the Coq system with a database of hints. Type inference compares types by calling the unification algorithm, that in turn looks into this database for solutions to problems that could not be solved otherwise. The user fills in the database by using a specific vernacular command to register the canonical solutions of his choice to some unification problems. Note that querying this database at unification time does not extend the trusted code base of the proof assistant. Just like the implicit arguments mechanism, this machinery is part of the proof engine and aims at decreasing the amount of type information provided by the user in order to describe a complete and well-formed Coq term.

In the current implementation the Canonical Structures database only stores solutions to unification problems of a very specific shape, that is the unification of a term with the projection of an unknown instance of a certain record type. This situation is typical of the issues faced when modeling algebraic structures with dependent record types. The following toy example illustrates how Canonical Structures have been used by their original authors. Suppose for instance that we have declared such a record type to define a naive interface for abelian (commutative) groups:

```
Structure abGrp : Type := AbGrp {
 carrier : Type;                zero : carrier;
 opp : carrier → carrier;       add : carrier → carrier → carrier;
 add_assoc : associative add;   add_comm : commutative add;
 zero_idl : left_id zero add;   add_oppl : left_inverse zero opp add }.
```

Here `carrier` is a projection extracting the type of the objects from a commutative group; similarly `zero` extracts the identity element, `add` the binary operation, ... We can prove the following theorem, valid for any instance of the structure:

```
Lemma subr0 : ∀(aG : abGrp) (x : carrier aG), add aG x (opp aG zero) = x
```

Now suppose that we have constructed an instance of this interface which equips the type `Z` of integers with a structure of commutative group:

```
Definition Z_abGrp := AbGrp Z Z0 Z1 Zopp Zadd ....
```

Despite this effort, there is no way to use lemma `subr0` to simplify an expression of the form `(Zadd z (Zopp Z0))`, with `(z : Z)`, since Coq's unification algorithm is not aware of the content of our library. More precisely Coq does not know that "`Z` forms a commutative group with `Zadd`, `Zopp` and `Z0`". The issue manifests itself as the difficult problem of unifying the type `Z` with `(carrier ?)`, where `?`

represents the unknown commutative group. But if we declare `Z_abGrp` as the canonical commutative group over `Z` using the following command

```
Canonical Structure Z_abGrp : abGrp.
```

the unification algorithm is able to solve that problem by filling the hole with `Z_abGrp`.

In the rest of the paper we describe how Canonical Structures works, why these seemingly atomic hints are applied iteratively and how to build an algebraic hierarchy exploiting this fact.

3 Type Inference and Unification

The rich type theory of the Coq system confers to type inference the power of computing values and proofs. Due to this richness, the type inference algorithm of Coq is way more involved than the ones of mainstream programming languages, even in presence of a type class mechanism à la Haskell [19]. Yet only a fragment of this algorithm plays a role in understanding how Canonical Structures work. The purpose of this section is to give a picture of the relevant fragment of Coq's type inference algorithm, obviously without ambitioning to provide a complete or fully formal exposition of type inference. Let us start by defining the common and simplified syntax of Coq terms and types we will be working with:

$$t ::= t\ t \mid \pi_n \mid r \mid x \mid ?x$$

Since binders do not play a role here we omit them in this syntax and only consider applicative terms. On the contrary projections (resp. constructors) of record types are central to the canonical structures mechanism and deserve to be identified explicitly, by π_i (resp. r). We also need names (x) to represent declared or defined terms and also symbols ($?x$) for unification variables.

An algorithm takes as input some terms and an environment Γ that collects declarations ($t : T \in \Gamma$), definitions ($x := t \in \Gamma$) and unification variable assignments ($?x := t \in \Gamma$). The output of an algorithm is of the same kind, where the resulting environment Γ' is obtained from Γ by possibly assigning some unification variables. We express the definition of an algorithm in relational style, as inference rules. A rule defining $\mathcal{R}$ has the following shape:

$$\frac{(\Gamma, t_1)\ \mathcal{R}\ (\Gamma', t_2) \qquad (\Gamma', t_2)\ \mathcal{R}\ (\Gamma'', t_3)}{(\Gamma, t_1)\ \mathcal{R}\ (\Gamma'', t_3)}\ \text{rule name}$$

One should consider arguments on the left of the $\mathcal{R}$ symbol as the input of the $\mathcal{R}$ algorithm while the ones on the right as the output. The premises of the rule, representing calls to the same or other algorithms, are always performed in left to right order. In all what follows, for sake of brevity, we omit the Γ in the rules.

Type inference, denoted by ":", is defined by the following two rules:

$$\frac{t : T \in \Gamma}{t : T}\ env \qquad \frac{t_1 : \forall x{:}A,B \qquad t_2 : A' \qquad A' \sim A}{t_1\ t_2 : B[x/t_2]}\ app$$

Type inference recursively traverses a term imposing that all atoms are declared in our implicit environment Γ (*env* rule) and that the type expected by the head of an application coincides with the one of its actual argument (*app* rule). Type inference performs this type comparison by calling the unification algorithm, denoted by "$\sim$" and defined by the following rules:

$$\frac{}{t \sim t}\ eq \qquad \frac{}{?x \sim t}\ assign \qquad \frac{t_1 \sim t_2}{t\ t_1 \sim t\ t_2}\ fst\text{-}order \qquad \frac{t_2 \rhd t_2' \qquad t_1 \sim t_2'}{t_1 \sim t_2}\ red$$

Unification succeeds on syntactically identical terms thanks to the rule *eq*. Unification variables are assigned to terms by the rule *assign*, under the usual occurrence check and type compatibility conditions that we omit for sake of brevity. The algorithm applies the *fst-order* rule whenever the head symbols of the two compared terms are identical. The *red* rule replaces the term t_2 by its reduced form t_2' before continuing the unification. The *fst-order* rule has precedence over the *red* rule. It goes without saying that the unification algorithm implemented by Coq features many more rules than this simplified version, including the symmetric rules of *red* and *assign*. But we omit these extra rules which play no role for the topic of this tutorial. The following set of five rules defines the reduction algorithm, denoted "$\rhd$":

$$\frac{x := t \in \Gamma}{x \rhd t}\ unfold \qquad \frac{?x := t \in \Gamma}{?x \rhd t}\ subst \qquad \frac{t \rhd r\ t_1\ \ldots\ t_n}{\pi_i\ t \rhd t_i}\ proj$$

$$\frac{t_1 \rhd t_1'}{t_1\ t \rhd t_1'\ t}\ hd\text{-}red \qquad \frac{t_1 \rhd t_1' \qquad t_1' \rhd t_1''}{t_1 \rhd t_1''}\ trans$$

Reduction can unfold global constants, by the *unfold* rule, and substitute assigned unification variables, by the *subst* rule, as well as reduce projections applied to record constructors, by the *proj* rule. Reduction is closed transitively (*trans* rule) and with respect to applicative contexts (*hd-red* rule).

4 Basic Overloading

In our first example we describe the infrastructure which creates an infix notation `==` that can be overloaded for several instances of binary comparisons. The "right" comparison function is chosen looking at the type of the compared objects. The user is required to declare a specific comparison function for each type of interest. The way of declaring such a function for a type is to build a dependent pair packaging together the type and the function and declare this pair as canonical.

Unfortunately this simple idea does not scale up properly: one often needs to attach to a type a bunch of operations (and properties) like in the abelian group example of section 2 and eventually reuse and extend the same set of operations later on, for example when defining a field. Hence the general pattern is to define a special package type, called a *class*, to describe the set of operations of interest. The user then builds an instance of the class by providing all the operations, and packages that instance together with the type for which these operations are the canonical ones.

Going back to our example, we begin by defining the `class` of objects that can be compared using `==`. This class just contains the comparison operation, named `cmp`, and is parametrized over the type `T` of the objects to be compared. The class is modeled using a `Record` whose constructor is named `Class`. We then define the package one has to build in order to use the `==` notation on a specific type. This is again modeled using a record that packages together a type, called `obj`, and an instance of the `class` just defined on the type `obj`. The commands `Structure` and `Record` are actually synonyms in Coq, but we consistently use `Structure` to define this last type of packages, given that their instances can be made canonical using the `Canonical Structure` command.[1]

We consistently use modules as name spaces, so that the short names like `class` get a qualifying prefix `EQ.` once the name space definition is finished. For sake of clarity, even if we comment the code of an open name space, we use the qualified versions of the names like `EQ.type` or `EQ.class`.

```
Module EQ.
  Record class (T : Type) := Class { cmp : T → T → Prop }.
  Structure type := Pack { obj : Type; class_of : class obj }.
  Definition op (e : type) : obj e → obj e → Prop :=
    let 'Pack _ (Class the_cmp) := e in the_cmp.
  Check op. (* ∀e : EQ.type, EQ.obj e → EQ.obj e → Prop *)
  Arguments op {e} x y : simpl never.
  Arguments Class {T} cmp.
```

The constant `(EQ.obj : EQ.type → Type)` is a projection of the `EQ.type` record. In order to access the comparison operator present in the nested `class` record, we define the `EQ.op` projection, whose type is displayed in the above code. Note that the first argument `(e : EQ.type)` of `op` is declared as an implicit one by the `Arguments` command.

In a `theory` name space we declare the set of notations and establish the bunch of properties that are shared by all the instances of the `EQ.type` structure, here a single infix notation `==` for the `EQ.op` operator. Since the first argument of `EQ.op` is implicit, this notation actually hides a hole standing for an unknown record of type `EQ.type` from which the comparison operator is extracted.

```
  Module theory.
  Notation "x == y" := (op x y) (at level 70).
  Check ∀(e : type) (a b : obj e), a == b.
  End theory.
End EQ.
Import EQ.theory.
Fail Check 3 == 3.
(* Error: The term "3" has type "nat"
   while it is expected to have type "EQ.obj ?1". *)
```

The `Check` command inside the `EQ.theory` name space verifies that we can use the infix notation to develop the theory for objects in the `EQ.type` named `e`.

[1] A more appropriate name would be `Canonical Instance` or simply `Canonical`.

After closing the `EQ.theory` module, type checking the expression `(3 == 3)` fails. To understand the error message we write the critical part of the execution of the type inference algorithm, boxing the unsolvable unification problem. We denote by τ the type of the `EQ.op` constant:

$$\tau := \forall e : \texttt{EQ.type}, \texttt{EQ.obj}\ e \to (\texttt{EQ.obj}\ e \to \texttt{Prop}) \in \Gamma$$

$$\dfrac{\dfrac{\dfrac{\texttt{EQ.op} : \tau \in \Gamma}{\texttt{EQ.op} : \tau} \qquad ?e : ?te \qquad \dfrac{}{?te \sim \texttt{EQ.type}}}{\texttt{EQ.op}\ ?e : \tau[e/?e]} \qquad 3 : \texttt{nat} \qquad \boxed{\texttt{nat} \sim \texttt{EQ.obj}\ ?e}}{(\texttt{EQ.op}\ ?e)\ 3 : \texttt{EQ.obj}\ ?e \to \texttt{Prop}}$$

We can see that the unification problem involves a projection of an unknown instance $?e$ of the structure, and that the algorithm cannot invent which such instance would project by `EQ.obj` on type `nat`. Coq will only manage to synthesize a closed term from the input `(3 == 3)` after some more work from the user. We first need to define an instance `nat_EQty` of the structure which contains the comparison operator `nat_eq` that we want to use for objects of type `nat`.

```
Definition nat_eq (x y : nat) := nat_compare x y = Eq.
Definition nat_EQcl : EQ.class nat := EQ.Class nat_eq.
Canonical Structure nat_EQty : EQ.type := EQ.Pack nat nat_EQcl.
Check 3 == 3. (* Works! *)
Eval compute in 3 == 4. (* Evaluates to Lt = Eq, indeed 3 ≠ 4. *)
```

We have moreover turned this definition into a canonical instance, via the `Canonical Structure` command: this extends the $\sim$ algorithm with the rule:

$$\dfrac{\texttt{nat} \sim \texttt{EQ.obj nat\_EQty} \qquad ?x \sim \texttt{nat\_EQty}}{\texttt{nat} \sim \texttt{EQ.obj}\ ?x} \tag{1}$$

This rule has two premises: the first one verifies that the `obj` component of the `nat_EQty` structure is `nat`, while the second one hints the solution for $?x$.

The following execution shows that the first premise is trivially satisfied:

$$\dfrac{\dfrac{\dfrac{\texttt{nat\_EQty} := \texttt{EQ.Pack nat nat\_EQcl} \in \Gamma}{\texttt{nat\_EQty} \rhd \texttt{EQ.Pack nat nat\_EQcl}}\ \textit{unfold}}{\texttt{EQ.obj nat\_EQty} \rhd \texttt{nat}}\ \textit{proj} \qquad \dfrac{}{\texttt{nat} \sim \texttt{nat}}\ \begin{matrix}\textit{eq}\\ \textit{red}\end{matrix}}{\texttt{nat} \sim \texttt{EQ.obj nat\_EQty}}$$

This reduction also shows that, once `nat_EQty` is assigned to $?x$, thanks to the *subst* rule the unification problem `nat` $\sim$ `EQ.obj` $?x$ is solved by simply reducing the right hand side to `nat`.

Actually, the hint generated by the `Canonical Structure` command is the following one, where the right hand side of the first premise is obtained by reducing the term (`EQ.obj nat_EQty`) into head normal form. This reduction is performed once and for all when the instance `nat_EQty` is made canonical.

$$\dfrac{\texttt{nat} \sim \texttt{nat} \qquad ?x \sim \texttt{nat\_EQty}}{\texttt{nat} \sim \texttt{EQ.obj}\ ?x} \tag{2}$$

Note that the first premise is not always trivial as this one. In the next example it plays the crucial role of iterating Canonical Structures resolution.

The last line of the Coq script shows that type inference has indeed found the right comparison function and can compute that `3` is different from `4`.

The next step is to be able to use the `==` on compound objects, like a pair of natural numbers. Hence we declare a *family* of hints providing a canonical method to compare pairs of objects when they both live in equality structures.

```
Fail Check ∀(e : EQ.type) (a b : EQ.obj e), (a,b) == (a,b).
(* Error: The term "(a, b)" has type "(EQ.obj e * EQ.obj e)"
   while it is expected to have type "EQ.obj ?15". *)
Definition pair_eq (e1 e2 : EQ.type) (x y : EQ.obj e1 * EQ.obj e2) :=
  fst x == fst y /\ snd x == snd y.
Definition pair_EQcl (e1 e2 : EQ.type) := EQ.Class (pair_eq e1 e2).
Canonical Structure pair_EQty (e1 e2 : EQ.type) : EQ.type :=
  EQ.Pack (EQ.obj e1 * EQ.obj e2) (pair_EQcl e1 e2).
```

We use the infix `*` to denote the type of pairs and we declare the obvious equality function over pairs as canonical. Note that `pair_EQty` has two parameters, `e1` and `e2`, of the same type. The following hint is added to the unification algorithm:

$$\frac{t_1 * t_2 \sim \texttt{EQ.obj}\ ?y * \texttt{EQ.obj}\ ?z \qquad ?x \sim \texttt{pair\_EQty}\ ?y\ ?z}{t_1 * t_2 \sim \texttt{EQ.obj}\ ?x} \tag{3}$$

Note that t_1 and t_2 are arbitrary terms, and hence the new rule is applicable whenever the unification problem involves the `EQ.obj` projection and the type constructor `*`. The unification variables $?y$ and $?z$ are fresh.

The following reduction justifies the shape of the first premise.

$$\dfrac{\dfrac{\dfrac{\texttt{pair\_EQty} := \texttt{EQ.Pack}\ \ldots \in \Gamma}{\texttt{pair\_EQty} \rhd \texttt{EQ.Pack}\ \ldots}\ \textit{unfold}}{\texttt{pair\_EQty}\ ?y\ ?z \rhd \texttt{EQ.Pack}\ (\texttt{EQ.obj}\ ?y * \texttt{EQ.obj}\ ?z)\ldots}\ \textit{hd-red}}{\texttt{EQ.obj}\ (\texttt{pair\_EQty}\ ?y\ ?z) \rhd \texttt{EQ.obj}\ ?y * \texttt{EQ.obj}\ ?z}\ \textit{proj}$$

It goes without saying that the first premise of (3) composes with the *fst-order* rule, given that both sides have the same head constant `*`. One could thus reformulate the hint in the following way, where the recursive calls on smaller, but similar, unification problems is evident:

$$\frac{t_1 \sim \texttt{EQ.obj}\ ?y \qquad t_2 \sim \texttt{EQ.obj}\ ?z \qquad ?x \sim \texttt{pair\_EQty}\ ?y\ ?z}{t_1 * t_2 \sim \texttt{EQ.obj}\ ?x} \tag{4}$$

This hint is correct by induction: if unification finds a value for $?y$ so that (`EQ.obj` $?y$) unifies with t_1 and a value for $?z$ so that (`EQ.obj` $?z$) unifies with t_2, then (`pair_EQty` $?y$ $?z$) is a solution for the problem, because it reduces to (`EQ.obj` $?y$ `*` `EQ.obj` $?z$) that unifies with (t_1 `*` t_2) by the *fst-order* rule.

It is now clear that type inference can make sense of a comparison by the `==` infix notation of any nested pairs of objects, as soon as their types are equipped

with registered equality structures, and that this could be made to work for other type constructors like `list`, `option`, etc...

5 Inheritance

In this section we show how to organize structures depending on one others by implementing inheritance. Let us start assuming we have defined a structure of types equipped with an order operator in a module named `LE`, exactly as we did for `EQ`, with an infix notation `<=`. As we equipped `nat` with an `EQ.type` canonical structure `nat_EQty` we also equip it with an `LE.type` canonical structure `nat_LEty`. Now we can mix the two operators `==` and `<=` on objects of type `nat`. This however is not sufficient for the development of the abstract theory shared by types that are instances of both structures:

```
Check 2 <= 3 /\ 2 == 2. (* Works! *)
Fail Check ∀(e : EQ.type) (x y : EQ.obj e), x <= y → y <= x → x == y.
(* Error: The term "x" has type "EQ.obj e"
   while it is expected to have type "LE.obj ?32". *)
```

In this failing example, the type of `x` and `y` is extracted from the equality structure, which provides no mean of guessing a related order structure. Using `LE.type` and `LE.obj` instead of `EQ.type` and `EQ.obj` would yield a similar error.

We need to craft a structure equipped with both comparison and order, and also impose that the two operations can be combined is a sensible way. We begin expressing this compatibility property. The packaging of this single property into a record is overkilling, but we aim at exposing the general schema here:

```
Module LEQ.
  Record mixin (e : EQ.type) (le : EQ.obj e → EQ.obj e → Prop) :=
    Mixin { compat : ∀(x y : EQ.obj e), (le x y /\ le y x) ↔ x == y }.
```

To express the property we need a type `e` of objects being comparable with `==` and an additional operation named `le`. The `mixin` is the only extra component we need to build the new class starting from the one of `EQ` and the one of `LE`.

```
  Record class T := Class {
    EQ_class : EQ.class T;
    LE_class : LE.class T;
    extra : mixin (EQ.Pack T EQ_class) (LE.cmp T LE_class) }.
  Structure type := _Pack { obj : Type; class_of : class obj }.
  Arguments Mixin {e le} _.
  Arguments Class {T} _ _ _.
```

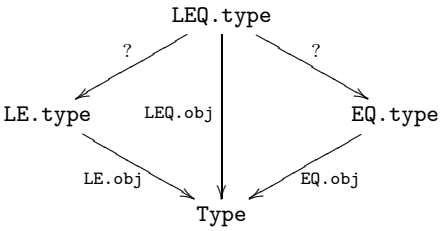
Some choices are arbitrary in this definition. For example one could use a symmetric mixin taking in input an `LE.type` and an operation `eq`. In general one parametrizes the mixin so to ease the writing of its content.

The reason why the `_Pack` constructor has been named this way becomes clear in the next section, where we implement a smarter constructor `Pack` to ease the declaration of canonical instances.

Unfortunately we have not yet fulfilled our goals completely:

```
Module theory.
Fail Check ∀(leq : type) (n m : obj leq), n <= m → n <= m → n == m.
(* Error: The term "n" has type "LEQ.obj leq"
   while it is expected to have type "LE.obj ?44". *)
```

The error message is better explained in the following graph where the nodes are types and the edges represent the inheritance relation. The two edges from `LEQ.type` to `EQ.type` and `LE.type` are missing: Coq does not know that an instance of the interface `LEQ.type` is also an instance of `LE.type` and `EQ.type`.



Again the issue manifests as an hard unification problem, that luckily falls in the domain of Canonical Structures. We only need to provide the following hints:

```
Definition to_EQ (leq : type) : EQ.type :=
  EQ.Pack (obj leq) (EQ_class _ (class_of leq)).
Definition to_LE (leq : type) : LE.type :=
  LE.Pack (obj leq) (LE_class _ (class_of leq)).
Canonical Structure to_EQ.
Canonical Structure to_LE.
```

The hint generated by the last command is the following one:

$$\frac{\texttt{LEQ.obj}\ t \sim \texttt{LEQ.obj}\ ?y \qquad ?x \sim \texttt{LEQ.to\_LE}\ ?y}{\texttt{LEQ.obj}\ t \sim \texttt{LE.obj}\ ?x} \tag{5}$$

The following reduction justifies the shape of the first premise and the correctness of the hinted solution:

$$\dfrac{\dfrac{\dfrac{\texttt{LEQ.to\_LE} := \texttt{LE.Pack}\ \ldots \in \Gamma}{\texttt{LEQ.to\_LE} \rhd \texttt{LE.Pack}\ \ldots}\ \mathit{unfold}}{\texttt{LEQ.to\_LE}\ ?y \rhd \texttt{LE.Pack}\ (\texttt{LEQ.obj}\ ?y)\ldots}\ \mathit{hd\text{-}red}}{\texttt{LE.obj}\ (\texttt{LEQ.to\_LE}\ ?y) \rhd \texttt{LEQ.obj}\ ?y}\ \mathit{proj}$$

Now we can state a lemma using both `==` and `<=` on an abstract type `leq`. In this simple case the proof is just the content of the mixin.

```
  Lemma lele_eq (leq : type) (x y : obj leq) : x <= y → y <= x → x == y
  Proof. ... Qed.
  Arguments lele_eq {leq} x y _ _.
  End theory.
End LEQ.
```

Even if objects of type `nat` can be compared with both `==` and `<=`, we did not prove that the corresponding operations are compatible. All we have to do is to build an instance of `LEQ.type` over `nat` and declare it as canonical.

```
Import LEQ.theory.
Example test1 (n m : nat) : n <= m → m <= n → n == m.
Proof. Fail apply (lele_eq n m). Abort.
(* Error: The term "n" has type "nat"
   while it is expected to have type "LEQ.obj ?48". *)
Lemma nat_LEQ_compat (n m : nat) : n <= m → m <= n → n == m.
Proof. ... Qed.
Definition nat_LEQmx := LEQ.Mixin nat_LEQ_compat.
Canonical Structure nat_LEQty : LEQ.type :=
  LEQ._Pack nat (LEQ.Class nat_EQcl nat_LEcl nat_LEQmx).
```

6 Proof Search

Just like in section 4, we can program a generic instance of equality-and-order structure for pairs of equality-and-order instances. Then the look-up for canonical instances can be iterated in a similar way, but this time for the equality, order and equality-and-order instances at once:

```
Lemma pair_LEQ_compat (l1 l2 : LEQ.type) (n m : LEQ.obj l1 * LEQ.obj l2):
  n <= m → m <= n → n == m.
Proof. ... Qed.
Definition pair_LEQmx (l1 l2 : LEQ.type) :=
  LEQ.Mixin (pair_LEQ_compat l1 l2).
Canonical Structure pair_LEQty (l1 l2 : LEQ.type) : LEQ.type :=
  LEQ._Pack (LEQ.obj l1 * LEQ.obj l2)
    (LEQ.Class
      (EQ.class_of (pair_EQty (to_EQ l1) (to_EQ l2)))
      (LE.class_of (pair_LEty (to_LE l1) (to_LE l2)))
      (pair_LEQmx l1 l2)).
Example test2 (n m : (nat * nat) * nat) : n <= m → m <= n → n == m.
Proof. now apply (lele_eq n m). Qed.
```

The proof of the toy example `test2` illustrates that the look-up for an instance of equality-and-order structure, which justifies the use of lemma `lele_eq`, truly amounts to a proof search mechanism. Indeed, neither have we ever programmed explicitly the operators equipping type `(nat * nat) * nat`, not have we proved the requirements of lemma `lele_eq` by hand on that instance: both the programs and the proofs have been synthesized from generic patterns.

7 Declaring Instances Made Easier

The declaration of the canonical structure `pair_LEQty` and `nat_LEQty` is still unsatisfactory: it is not only very verbose, but also very redundant. Indeed we had

to provide by hand many components the system is in principle able to infer. In particular the `EQ.type` and `LE.type` structures for `nat` and the pair type are provided explicitly by hand even if we have registered them as canonical. In this section we show how to program a packager that given the type and the mixin that characterizes the `LEQ.type` we are building, infers all the remaining fields.

The main difficulty is that the information registered in the Canonical Structures database can be retrieved only when a precise problem is posed to the unification algorithm. We hence start with the description of some generic Swiss knife which allows to pose arbitrary problems to the unification algorithm and to extract the resulting information.

The first problem to overcome is that unification (and its Canonical Structures database) is used by type inference to process *types*, while in many occasions we want to enforce the unification of *terms*.[2] However we can inject values into types, even artificially, using the dependent types of Coq. To this purpose we define and use a `phantom` type. This construction is also used in the programming language context to trick the type system into enforcing extra invariants [11].

```
Module infrastructure.
  Inductive phantom {T : Type} (t : T) : Type := Phantom.
  Definition unify {T1 T2} (t1 : T1) (t2 : T2) (s : option string) :=
    phantom t1 → phantom t2.
  Definition id {T} {t : T} (x : phantom t) := x.
  Notation "[find v | t1 ∼ t2 ] rest" :=
    (fun v (_ : unify t1 t2 None) => rest) ...
  Notation "[find v | t1 ∼ t2 | msg ] rest" :=
    (fun v (_ : unify t1 t2 (Some msg)) => rest) ...
  Notation "'Error: t msg" := (unify _ t (Some msg)) ...
End infrastructure.
```

Note that in order to improve the error messages, and to facilitate debugging, the `unify` function optionally holds a string representing an error message. The notation "`[find v | t1 ∼ t2 | msg ] rest`" should be read as: "find `v` such that `t1` unifies with `t2` or fail with `msg`", then continue with `rest`.

The second problem is that, once terms are lifted to the types level, we still need to be able to call the unification procedure *at the right moment*. The infrastructure we want to build has to be generic, i.e. be expressed for an arbitrary type `T` and mixin `m`, but there is no canonical structure for an arbitrary type `T`. We need to run the unification procedure only when `T` is provided. To this extent the terms to be unified are stored as `unify t1 t2` that is defined as `phantom t1 → phantom t2`.[3] Each argument of type `unify` will be later instantiated with an identity function, whose type `phantom t → phantom t` (for some `t`) is required to match `phantom t1 → phantom t2`, and hence forces the unification of `t1` with `t2`.

[2] It is avoidable in this context, but we prefer to present this mechanism in its full generality given its ubiquity in the Mathematical Components library.

[3] On the contrary using `t1 = t2` would force their type to be immediately unifiable.

With this infrastructure we can define a packager to build an inhabitant `(LEQ._Pack T (LEQ.Class ce co m))` of the `LEQ.type` record given the type for the objects `T` and the mixin `m0`.

```
Import infrastructure.
Definition packager T e0 le0 (m0 : LEQ.mixin e0 le0) :=
    [find e  | EQ.obj e ~ T       | "is not an EQ.type" ]
    [find o  | LE.obj o ~ T       | "is not an LE.type" ]
    [find ce | EQ.class_of e ~ ce ]
    [find co | LE.class_of o ~ co ]
    [find m  | m ~ m0             | "is not the right mixin" ]
  LEQ._Pack T (LEQ.Class ce co m).
Notation Pack T m := (packager T _ _ m _ id _ id _ id _ id).
```

The parameters `e0` and `le0` are needed only for the argument `m0` to be well-typed. Now remark that this packager eventually builds an instance of the `LEQ.type` structure from the input type `T` plus three components `ce`, `co` and `m` that are not calculated from the input mixin `m0` by a Coq function. The component `ce` (resp. `co`) is the class field of the record `e` (resp. `o`), which is itself an `EQ.type` (resp. `LE.type`) that is inferred as the canonical instance over `T`. Finally `m` is forced to be equal to the input mixin `m0`. The `Pack` notation is just a macro to provide the packager with enough unknown values `_` and enough identity functions for the unification problems to be triggered *when the notation is used and* `T` *is provided*. Canonical instance declarations can then be shortened as follows:

```
Canonical Structure nat_LEQty := Eval hnf in Pack nat nat_LEQmx.
Canonical Structure pair_LEQty (l1 l2 : LEQ.type) :=
  Eval hnf in Pack (LEQ.obj l1 * LEQ.obj l2) (pair_LEQmx l1 l2).
```

When we declare the first canonical instance, type inference gives a type to `(Pack nat nat_LEQmx)`. All the `_` part of the `Pack` notation are inferred by the unification algorithm that in turn finds a value for `e`, then `o`, then `ce`, then `co` and finally `m`. The "`Eval hnf in`" command reduces away the abstractions in the body of the packager. Error reporting is also quite accurate:

```
Fail Canonical Structure e := Eval hnf in Pack bool nat_LEQmx.
(* ... 'Error: bool "is not an EQ.type" *)
Fail Canonical Structure e := Eval hnf in Pack (nat * nat) nat_LEQmx.
(* ... 'Error: nat_LEQmx "is not the right mixin" *)
```

Note that, in this simple case, the packager could be simplified by using explicitly the `e0` parameter of the `m0` mixin for the value of the `e` component.

8 Conclusions and Related Works

As a conclusion we compare three similar mechanisms that have been implemented independently in proof assistants based on the Calculus of Inductive Constructions: Coq's Type Classes [15], which have also been used to develop

hierarchies of algebraic structures [16]; Coq's Canonical Structures [14], which we have presented in this tutorial; and Matita's Unification Hints [1], that can be seen as a generalization of Canonical Structures.

	Type Classes	Canonical Structures	Unification Hints
1^{st} class	Yes	Yes	Yes
local instance	Yes	Almost	Yes
search engine	ad hoc	unification	unification
priority/overlap	explicit	encodable	explicit
backtracking	native	encodable	/
package/inheritance	unbundled/trivial	bundled/easy	bundled/easy
narrowing	/	No	Yes

1^{st} class: With all these mechanisms, interfaces are record types, hence first class objects. This is fundamental to build new types and instances on the fly in the middle of proofs — one cannot for instance declare a module in the middle of a proof. This is also crucial to develop a generic theory by quantifying on all the instances of such a structures to express statements like "the first order theory of an algebraically closed field has the quantifier elimination property" [3].

Local Instance: Even if new structure instances can be built in the middle of a proof, Canonical Structures require all the building blocks to be globally defined. In other words there is no support for hinting the unification using a construction one obtains in the middle of a proof. On the contrary Type Classes take into account local instances available in the proof context. Actually one can program unification to postpone all unification problems that fail because of a missing Canonical Structure in the following way:

```
Canonical Structure failsafe t f : EQ.type := EQ.Pack t (EQ.Class f).
Set Printing All.
Check (true == true). (* @EQ.op (failsafe bool ?9) true true : Prop *)
Fail Lemma test : true == true.
(* Cannot infer an internal placeholder of type "bool → bool → Prop *)
```

The `failsafe` Canonical Structure has a special status, given that its `obj` component `t` is a parameter. Canonical Structures of this kind are called "default" and are used only if no other canonical structure can be applied. With this hint type inference can give a type to `(true == true)` even if there is no `EQ.type` declared as canonical for the type `bool`. Unfortunately Coq 8.4 handles unresolved unification variables like `?9` in a non uniform way. For example a statement of a theorem cannot contain unresolved variables.

Search Engine: This is certainly the major limitation of all three mechanisms, that, to our knowledge, are equally undocumented for that respect. To understand failures one may need to dig into the sources of the systems.

Backtracking and Priorities: Backtracking is a native mechanism of Type Classes' search engine. This mechanism offers vernacular commands to assign weighted priorities to the instances declared by the user. Instances can overlap

and the search engine may try all possibilities. Instances of Canonical Structures cannot overlap. However, one can alias a term by means of a definition and encode priorities into a chain of aliases (See [10], section 2.3). Instances of Unification Hints can overlap, and they are tried in an order provided explicitly by the user. The language for the declaration of Unification Hints has a syntax very close to the one we used in this paper to explain hint (4) in section 4 and a notion of cut à la Prolog would clearly apply but is not currently implemented.

Package/Inheritance: Different mechanisms impose different styles to declare the interfaces. Type Classes enforce the unbundled approach, where the values used to search instances have to be parameters of the structure. Canonical Structures and Unification Hints enforce a bundled (also called packed) approach, where the values used to search for instances have to be fields of the structure. Implementing inheritance relation between interfaces is possible using all three mechanisms. The tricky case is the one of multiple inheritance like our `LEQ` of section 5, where a structure has to inherit from two a priori unrelated structures that nonetheless provide operators and specifications on the same type. This problem is trivial with unbundled structures, as the common domain type constraint can be expressed syntactically. With structures expressed in bundled style the best solution known is the one of packed-classes [5] we used in section 5, that provides a general solution to single and multiple inheritance. This solution is used consistently in the whole web of interfaces defined in the Mathematical Components libraries.

Narrowing: Type Classes do not extend unification, so this point does not apply. Canonical Structures extend unification only when dealing with problems involving a record projection. Unification Hints let the user extend unification more freely. For example a unification hint could enable the unification algorithm to solve the problem "?x `* (S n)` $\sim$ `0`" by assigning ?x to `0`.

A very natural question is if it is possible to mix Canonical Structures and Type Classes in Coq. The answer is yes. For example one can build an `EQ.type` instance using the Type Classes search engine as follows:

```
Existing Class EQ.class.
Canonical Structure failsafe T {c : EQ.class T} : EQ.type := EQ.Pack T c.
Instance bool_EQcl : EQ.class bool := EQ.Class bool bool_cmp.
Check true == true. (* Works! ?9 assigned to bool_EQcl *)
```

The converse is also possible.

```
Class eq_class {A} : Type := Class { cmp : A → A → Prop }.
Notation "x === y" := (cmp x y) (at level 70).
Instance find_CS (e : EQ.type) : eq_class := Class (EQ.obj e) (@EQ.op e).
Set Printing All.
Check 0 === 0. (* @cmp nat (find_CS nat_EQty) O O : Prop *)
```

Acknowledgments. We are grateful to Frédéric Chyzak, Georges Gonthier, Laurence Rideau and Matthieu Sozeau for proofreading this text.

References

[1] Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: Hints in unification. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 84–98. Springer, Heidelberg (2009)

[2] 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)

[3] Cohen, C.: Formalized algebraic numbers: construction and first order theory. PhD thesis, École polytechnique (2012)

[4] Garillot, F.: Generic Proof Tools and Finite Group Theory. PhD thesis, École polytechnique (2011)

[5] 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)

[6] Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in coq. J. Symb. Comput. 34(4), 271–286 (2002)

[7] Gonthier, G.: Point-free, set-free concrete linear algebra. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 103–118. Springer, Heidelberg (2011)

[8] 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)

[9] Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA (2012)

[10] Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ICFP, pp. 163–175. ACM (2011)

[11] Hinze, R.: Fun with phantom types. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, Cornerstones of Computing, pp. 245–262 (2003)

[12] Huet, G.P., Saïbi, A.: Constructive category theory. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction, pp. 239–276. The MIT Press (2000)

[13] Saïbi, A.: Typing algorithm in type theory with inheritance. In: Lee, P., Henglein, F., Jones, N.D. (eds.) POPL, pp. 292–301. ACM Press (1997)

[14] Saïbi, A.: Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types: application à la Théorie des Catégories. PhD thesis, Université Paris VI (1999)

[15] 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)

[16] Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science 21(4), 795–825 (2011)

[17] The Coq Development Team. The Coq proof assistant, version 8.4., `http://coq.inria.fr`

[18] The Mathematical Component Team. A Formalization of the Odd Order Theorem using the Coq proof assistant (September 2012), `http://www.msr-inria.inria.fr/Projects/math-components/feit-thompson`

[19] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of POPL, pp. 60–76. ACM (1989)

MaSh: Machine Learning for Sledgehammer

Daniel Kühlwein[1], Jasmin Christian Blanchette[2], Cezary Kaliszyk[3], and Josef Urban[1]

[1] ICIS, Radboud Universiteit Nijmegen, The Netherlands
[2] Fakultät für Informatik, Technische Universität München, Germany
[3] Institut für Informatik, Universität Innsbruck, Austria

In memoriam Piotr Rudnicki 1951–2012

Abstract. Sledgehammer integrates automatic theorem provers in the proof assistant Isabelle/HOL. A key component, the relevance filter, heuristically ranks the thousands of facts available and selects a subset, based on syntactic similarity to the current goal. We introduce MaSh, an alternative that learns from successful proofs. New challenges arose from our "zero-click" vision: MaSh should integrate seamlessly with the users' workflow, so that they benefit from machine learning without having to install software, set up servers, or guide the learning. The underlying machinery draws on recent research in the context of Mizar and HOL Light, with a number of enhancements. MaSh outperforms the old relevance filter on large formalizations, and a particularly strong filter is obtained by combining the two filters.

1 Introduction

Sledgehammer [27] is a subsystem of the proof assistant Isabelle/HOL [25] that discharges interactive goals by harnessing external automatic theorem provers (ATPs). It heuristically selects a number of relevant facts (axioms, definitions, or lemmas) from the thousands available in background libraries and the user's formalization, translates the problem to the external provers' logics, and reconstructs any machine-found proof in Isabelle (Sect. 2). The tool is popular with both novices and experts.

Various aspects of Sledgehammer have been improved since its introduction, notably the addition of SMT solvers [7], the use of sound translation schemes [8], close cooperation with the first-order superposition prover SPASS [9], and of course advances in the underlying provers themselves. Together, these enhancements increased the success rate from 48% to 64% on the representative "Judgment Day" benchmark suite [9, 10].

One key component that has received little attention is the relevance filter. Meng and Paulson [23] designed a filter, MePo, that iteratively ranks and selects facts similar to the current goal, based on the symbols they contain. Despite its simplicity, and despite advances in prover technology [9, 14, 30], this filter greatly increases the success rate: Most provers cannot cope with tens of thousands of formulas, and translating so many formulas would also put a heavy burden on Sledgehammer. Moreover, the translation of Isabelle's higher-order constructs and types is optimized globally for a problem—smaller problems make more optimizations possible, which helps the automatic provers.

Coinciding with the development of Sledgehammer and MePo, a line of research has focused on applying machine learning to large-theory reasoning. Much of this work

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 35–50, 2013.

has been done on the vast Mizar Mathematical Library (MML) [1], either in its original Mizar [22] formulation or in first-order form as the Mizar Problems for Theorem Proving (MPTP) [33]. The MaLARea system [34,38] and the competitions CASC LTB and Mizar@Turing [31] have been important milestones. Recently, comparative studies involving MPTP [2, 20] and the Flyspeck project in HOL Light [17] have found that fact selectors based on machine learning outperform symbol-based approaches.

Several learning-based advisors have been implemented and have made an impact on the automated reasoning community. In this paper, we describe a tool that aims to bring the fruits of this research to the Isabelle community. This tool, MaSh, offers a memoryful alternative to MePo by learning from successful proofs, whether human-written or machine-generated.

Sledgehammer is a one-click technology—fact selection, translation, and reconstruction are fully automatic. For MaSh, we had four main design goals:

- *Zero-configuration*: The tool should require no installation or configuration steps, even for use with unofficial repository versions of Isabelle.
- *Zero-click*: Existing users of Sledgehammer should benefit from machine learning, both for standard theories and for their custom developments, without having to change their workflow.
- *Zero-maintenance*: The tool should not add to the maintenance burden of Isabelle. In particular, it should not require maintaining a server or a database.
- *Zero-overhead*: Machine learning should incur no overhead to those Isabelle users who do not employ Sledgehammer.

By pursuing these "four zeros," we hope to reach as many users as possible and keep them for as long as possible. These goals have produced many new challenges.

MaSh's heart is a Python program that implements a custom version of a weighted sparse naive Bayes algorithm that is faster than the naive Bayes algorithm implemented in the SNoW [12] system used in previous studies (Sect. 3). The program maintains a persistent state and supports incremental, nonmonotonic updates. Although distributed with Isabelle, it is fully independent and could be used by other proof assistants, automatic theorem provers, or applications with similar requirements.

This Python program is used within a Standard ML module that integrates machine learning with Isabelle (Sect. 4). When Sledgehammer is invoked, it exports new facts and their proofs to the machine learner and queries it to obtain relevant facts. The main technical difficulty is to perform the learning in a fast and robust way without interfering with other activities of the proof assistant. Power users can enhance the learning by letting external provers run for hours on libraries, searching for simpler proofs.

A particularly strong filter, MeSh, is obtained by combining MePo and MaSh. The three filters are compared on large formalizations covering the traditional application areas of Isabelle: cryptography, programming languages, and mathematics (Sect. 5). These empirical results are complemented by Judgment Day, a benchmark suite that has tracked Sledgehammer's development since 2010. Performance varies greatly depending on the application area and on how much has been learned, but even with little learning MeSh emerges as a strong leader.

2 Sledgehammer and MePo

Whenever Sledgehammer is invoked on a goal, the MePo (*Me*ng–*P*aulso*n*) filter selects n facts $\phi_1, \ldots, \phi_n$ from the thousands available, ordering them by decreasing estimated relevance. The filter keeps track of a set of relevant symbols—i.e., (higher-order) constants and fixed variables—initially consisting of all the goal's symbols. It performs the following steps iteratively, until n facts have been selected:

1. Compute each fact's score, as roughly given by $r/(r+i)$, where r is the number of relevant symbols and i the number of irrelevant symbols occurring in the fact.
2. Select all facts with perfect scores as well as some of the remaining top-scoring facts, and add all their symbols to the set of relevant symbols.

The implementation refines this approach in several ways. Chained facts (inserted into the goal by means of the keywords `using`, `from`, `then`, `hence`, and `thus`) take absolute priority; local facts are preferred to global ones; first-order facts are preferred to higher-order ones; rare symbols are weighted more heavily than common ones; and so on.

MePo tends to perform best on goals that contain some rare symbols; if all the symbols are common, it discriminates poorly among the hundreds of facts that could be relevant. There is also the issue of starvation: The filter, with its iterative expansion of the set of relevant symbols, effectively performs a best-first search in a tree and may therefore ignore some relevant facts close to the tree's root.

The automatic provers are given prefixes $\phi_1, \ldots, \phi_m$ of the selected n facts. The order of the facts—the estimated relevance—is exploited by some provers to guide the search. Although Sledgehammer's default time limit is 30 s, the automatic provers are invoked repeatedly for shorter time periods, with different options and different number of facts $m \leq n$; for example, SPASS is given as few as 50 facts in some slices and as many as 1000 in others. Excluding some facts restricts the search space, helping the prover find deeper proofs within the allotted time, but it also makes fewer proofs possible.

The supported ATP systems include the first-order provers E [29], SPASS [9], and Vampire [28]; the SMT solvers CVC3 [4], Yices [13], and Z3 [24]; and the higher-order provers LEO-II [5] and Satallax [11].

Once a proof is found, Sledgehammer minimizes it by invoking the prover repeatedly with subsets of the facts it refers to. The proof is then reconstructed in Isabelle by a suitable proof text, typically a call to the built-in resolution prover Metis [16].

Example 1. Given the goal

$$\mathsf{map}\ f\ xs = ys \Longrightarrow \mathsf{zip}\ (\mathsf{rev}\ xs)\ (\mathsf{rev}\ ys) = \mathsf{rev}\ (\mathsf{zip}\ xs\ ys)$$

MePo selects 1000 facts: *rev_map*, *rev_rev_ident*, …, *add_numeral_special*(3). The prover E, among others, quickly finds a minimal proof involving the 4th and 16th facts:

$$\begin{aligned} \textit{zip\_rev}\colon\ & \mathsf{length}\ xs = \mathsf{length}\ ys \Longrightarrow \mathsf{zip}\ (\mathsf{rev}\ xs)\ (\mathsf{rev}\ ys) = \mathsf{rev}\ (\mathsf{zip}\ xs\ ys) \\ \textit{length\_map}\colon\ & \mathsf{length}\ (\mathsf{map}\ f\ xs) = \mathsf{length}\ xs \end{aligned}$$

Example 2. MePo's tendency to starve out useful facts is illustrated by the following goal, taken from Paulson's verification of cryptographic protocols [26]:

$$\mathsf{used}\ [] \subseteq \mathsf{used}\ \mathit{evs}$$

A straightforward proof relies on these four lemmas:

$$\begin{aligned} \textit{used\_Nil}&: \mathsf{used}\ [] = \bigcup_B \mathsf{parts}\ (\mathsf{initState}\ B) \\ \textit{initState\_into\_used}&: X \in \mathsf{parts}\ (\mathsf{initState}\ B) \Longrightarrow X \in \mathsf{used}\ \textit{evs} \\ \textit{subsetI}&: (\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B \\ \textit{UN\_iff}&: b \in \bigcup_{x \in A} B\ x \longleftrightarrow \exists x \in A.\ b \in B\ x \end{aligned}$$

The first two lemmas are ranked 6807th and 6808th, due to the many initially irrelevant constants ($\bigcup$, parts, initState, and $\in$). In contrast, all four lemmas appear among MaSh's first 45 facts and MeSh's first 77 facts.

3 The Machine Learning Engine

MaSh (*Ma*chine Learning for *S*ledge*h*ammer) is a Python program for fact selection with machine learning.[1] Its default learning algorithm is an approximation of naive Bayes adapted to fact selection. MaSh can perform fast model updates, overwrite data points, and predict the relevance of each fact. The program can also use the much slower naive Bayes algorithm implemented by SNoW [12].

3.1 Basic Concepts

MaSh manipulates theorem proving concepts such as facts and proofs in an agnostic way, as "abstract nonsense":

- A *fact* ϕ is a string.
- A *feature* f is also a string. A positive *weight* w is attached to each feature.
- *Visibility* is a partial order $\prec$ on facts. A fact ϕ is visible *from* a fact ϕ' if $\phi \prec \phi'$, and visible *through* the set of facts Φ if there exists a fact $\phi' \in \Phi$ such that $\phi \preceq \phi'$.
- The *parents* of a fact are its (immediate) predecessors with respect to $\prec$.
- A *proof* Π for ϕ is a set of facts visible from ϕ.

Facts are described abstractly by their feature sets. The features may for example be the symbols occurring in a fact's statement. Machine learning proceeds from the hypothesis that facts with similar features are likely to have similar proofs.

3.2 Input and Output

MaSh starts by loading the persistent model (if any), executes a list of commands, and saves the resulting model on disk. The commands and their arguments are

`learn` *fact parents features proof*
`relearn` *fact proof*
`query` *parents features hints*

The `learn` command teaches MaSh a new fact ϕ and its proof Π. The parents specify how to extend the visibility relation for ϕ, and the features describe ϕ. In addition to the supplied proof $\Pi \vdash \phi$, MaSh learns the trivial proof $\phi \vdash \phi$; hence something is learned even if $\Pi = \emptyset$ (which can indicate that no suitable proof is available).

[1] The source code is distributed with Isabelle2013 in the directory `src/HOL/Tools/Sledgehammer/MaSh/src`.

The `relearn` command forgets a fact's proof and learns a new one.

The `query` command ranks all facts visible through the given parents by their predicted relevance with respect to the specified features. The optional hints are facts that guide the search. MaSh temporarily updates the model with the hints as a proof for the current goal before executing the query.

The commands have various preconditions. For example, for `learn`, ϕ must be fresh, the parents must exist, and all facts in Π must be visible through the parents.

3.3 The Learning Algorithm

MaSh's default machine learning algorithm is a weighted version of sparse naive Bayes. It ranks each visible fact ϕ as follows. Consider a `query` command with the features $f_1, \ldots, f_n$ weighted $w_1, \ldots, w_n$, respectively. Let P denote the number of proofs in which ϕ occurs, and $p_j \leq P$ the number of such proofs associated with facts described by f_j (among other features). Let π and σ be predefined weights for known and unknown features, respectively. The estimated relevance is given by

$$r(\phi, f_1, \ldots, f_n) = \ln P + \sum_{j\colon p_j \neq 0} w_j \big(\ln(\pi p_j) - \ln P\big) + \sum_{j\colon p_j = 0} w_j \sigma$$

When a fact is learned, the values for P and p_j are initialized to a predefined weight τ. The models depend only on the values of P, p_j, π, σ, and τ, which are stored in dictionaries for fast access. Computing the relevance is faster than with standard naive Bayes because only the features that describe the current goal need to be considered, as opposed to all features (of which there may be tens of thousands). Experiments have found the values $\pi = 10$, $\sigma = -15$, and $\tau = 20$ suitable.

A crucial technical issue is to represent the visibility relation efficiently as part of the persistent state. Storing all the ancestors for each fact results in huge files that must be loaded and saved, and storing only the parents results in repeated traversals of long parentage chains to obtain all visible facts. MaSh solves this dilemma by complementing parentage with a cache that stores the ancestry of up to 100 recently looked-up facts. The cache not only speeds up the lookup for the cached facts but also helps shortcut the parentage chain traversal for their descendants.

4 Integration in Sledgehammer

Sledgehammer's MaSh-based relevance filter is implemented in Standard ML, like most of Isabelle.[2] It relies on MaSh to provide suggestions for relevant facts whenever the user invokes Sledgehammer on an interactive goal.

4.1 The Low-Level Learner Interface

Communication with MaSh is encapsulated by four ML functions. The first function resets the persistent state; the last three invoke MaSh with a list of commands:

[2] The code is located in Isabelle2013's files `src/HOL/Tools/Sledgehammer/sledgehammer_mash.ML`, `src/HOL/TPTP/mash_export.ML`, and `src/HOL/TPTP/mash_eval.ML`.

$$\begin{aligned}
&\mathsf{MaSh.unlearn}\ ()\\
&\mathsf{MaSh.learn}\ [(\mathit{fact}_1, \mathit{parents}_1, \mathit{features}_1, \mathit{proof}_1), \ldots,\\
&\qquad\qquad (\mathit{fact}_n, \mathit{parents}_n, \mathit{features}_n, \mathit{proof}_n)]\\
&\mathsf{MaSh.relearn}\ [(\mathit{fact}_1, \mathit{proof}_1), \ldots, (\mathit{fact}_n, \mathit{proof}_n)]\\
&\mathit{suggestions} = \mathsf{MaSh.query}\ \mathit{parents}\ \mathit{features}\ \mathit{hints}
\end{aligned}$$

To track what has been learned and avoid violating MaSh's preconditions, Sledgehammer maintains its own persistent state, mirrored in memory. This mainly consists of the visibility graph, a directed acyclic graph whose vertices are the facts known to MaSh and whose edges connect the facts to their parents. (MaSh itself maintains a visibility graph based on `learn` commands.) The state is accessed via three ML functions that use a lock to guard against race conditions in a multithreaded environment [41] and keep the transient and persistent states synchronized.

4.2 Learning from and for Isabelle

Facts, features, proofs, and visibility were introduced in Sect. 3.1 as empty shells. The integration with Isabelle fills these concepts with content.

Facts. Communication with MaSh requires a string representation of Isabelle facts. Each theorem in Isabelle carries a stable "name hint" that is identical or very similar to its fully qualified user-visible name (e.g., *List.map.simps_2* vs. *List.map.simps*(2)). Top-level lemmas have unambiguous names. Local facts in a structured Isar proof [40] are disambiguated by appending the fact's statement to its name.

Features. Machine learning operates not on the formulas directly but on sets of features. The simplest scheme is to encode each symbol occurring in a formula as its own feature. The experience with MePo is that other factors help—for example, the formula's types and type classes or the theory it belongs to. The MML and Flyspeck evaluations revealed that it is also helpful to preserve parts of the formula's structure, such as subterms [3, 17].

Inspired by these precursors, we devised the following scheme. For each term in the formula, excluding the outer quantifiers, connectives, and equality, the features are derived from the nontrivial first-order patterns up to a given depth. Variables are replaced by the wildcard _ (underscore). Given a maximum depth of 2, the term $\mathsf{g}\ (\mathsf{h}\ x\ \mathsf{a})$, where constants g, h, a originate from theories T, U, V, yields the patterns

$$\mathsf{T.g(\_)} \qquad \mathsf{T.g(U.h(\_,\_))} \qquad \mathsf{U.h(\_,\_)} \qquad \mathsf{U.h(\_,V.a)} \qquad \mathsf{V.a}$$

which are simplified and encoded into the features

$$\mathsf{T.g} \qquad \mathsf{T.g(U.h)} \qquad \mathsf{U.h} \qquad \mathsf{U.h(V.a)} \qquad \mathsf{V.a}$$

Types, excluding those of propositions, Booleans, and functions, are encoded using an analogous scheme. Type variables constrained by type classes give rise to features corresponding to the specified type classes and their superclasses. Finally, various pieces of metainformation are encoded as features: the theory to which the fact belongs; the kind of rule (e.g., introduction, simplification); whether the fact is local; whether the formula contains any existential quantifiers or λ-abstractions.

Guided by experiments similar to those of Sect. 5, we attributed the following weights to the feature classes:

Fixed variable	20	Type	2	Presence of $\exists$	2
Constant	16	Theory	2	Presence of λ	2
Localness	8	Kind of rule	2	Type class	1

Example 3. The lemma $\mathsf{transpose}\ (\mathsf{map}\ (\mathsf{map}\ f)\ xs) = \mathsf{map}\ (\mathsf{map}\ f)\ (\mathsf{transpose}\ xs)$ from the *List* theory has the following features and weights (indicated by subscripts):

List_2	$\mathsf{List.transpose}_{16}$
$\mathsf{List.list}_2$	$\mathsf{List.transpose(List.map)}_{16}$
$\mathsf{List.map}_{16}$	$\mathsf{List.map(List.transpose)}_{16}$
$\mathsf{List.map(List.map)}_{16}$	$\mathsf{List.map(List.map, List.transpose)}_{16}$

Proofs. MaSh predicts which facts are useful for proving the goal at hand by studying successful proofs. There is an obvious source of successful proofs: All the facts in the loaded theories are accompanied by proof terms that store the dependencies [6]. However, not all available facts are equally suitable for learning. Many of them are derived automatically by definitional packages (e.g., for inductive predicates, datatypes, recursive functions) and proved using custom tactics, and there is not much to learn from those highly technical lemmas. The most interesting lemmas are those stated and proved by humans. Slightly abusing terminology, we call these "Isar proofs."

Even for user lemmas, the proof terms are overwhelmed by basic facts about the logic, which are tautologies in their translated form. Fortunately, these tautologies are easy to detect, since they contain only logical symbols (equality, connectives, and quantifiers). The proofs are also polluted by decision procedures; an extreme example is the Presburger arithmetic procedure, which routinely pulls in over 200 dependencies. Proofs involving over 20 facts are considered unsuitable and simply ignored.

Human-written Isar proofs are abundant, but they are not necessarily the best raw material to learn from. They tend to involve more, different facts than Sledgehammer proofs. Sometimes they rely on induction, which is difficult for automatic provers; but even excluding induction, there is evidence that the provers work better if the learned proofs were produced by similar provers [20, 21].

A special mode of Sledgehammer runs an automatic prover on all available facts to learn from ATP-generated proofs. Users can let it run for hours at at time on their favorite theories. The Isar proof facts are passed to the provers together with a few dozens of MePo-selected facts. Whenever a prover succeeds, MaSh discards the Isar proof and learns the new minimized proof (using MaSh.relearn). Facts with large Isar proofs are processed first since they stand to gain the most from shorter proofs.

Visibility. The loaded background theories and the user's formalization, including local lemmas, appear to Sledgehammer as a vast collection of facts. Each fact is tagged with its own abstract theory value, of type theory in ML, that captures the state of affairs when it was introduced. Sledgehammer constructs the visibility graph by using the (very fast) subsumption order $\trianglelefteq$ on theory.

A complication arises because $\trianglelefteq$ lifted to facts is a preorder, whereas the graph must encode a partial order $\preceq$. Antisymmetry is violated when facts are registered together.

Despite the simultaneity, one fact's proof may depend on another's; for example, an inductive predicate's definition *p_def* is used to derive introduction and elimination rules *pI* and *pE*, and yet they may share the same theory. Hence, some care is needed when constructing $\preceq$ from $\trianglelefteq$ to ensure that *p_def* $\preceq$ *pI* and *p_def* $\preceq$ *pE*.

When performing a query, Sledgehammer needs to compute the current goal's parents. This involves finding the maximal vertices of the visibility graph restricted to the facts available in the current Isabelle proof context. The computation is efficient for graphs with a quasi-linear structure, such as those that arise from Isabelle theories: Typically, only the first fact of a theory will have more than one parent. A similar computation is necessary when teaching MaSh new facts.

4.3 Relevance Filters: MaSh and MeSh

Sledgehammer's MaSh-based relevance filter computes the current goal's parents and features; then it queries the learner program (using MaSh.query), passing the chained facts as hints. This process usually takes about one second on modern hardware, which is reasonable for a tool that may run for half a minute. The result is a list with as many suggestions as desired, ordered by decreasing estimated relevance.

Relying purely on MaSh for relevance filtering raises an issue: MaSh may not have learned all the available facts. In particular, it will be oblivious to the very latest facts, introduced after Sledgehammer was invoked for the last time, and these are likely to be crucial for the proof. The solution is to enrich the raw MaSh data with a proximity filter, which sorts the available facts by decreasing proximity in the proof text.

Instead of a plain linear combination of ranks, the enriched MaSh filter transforms ranks into probabilities and takes their weighted average, with weight 0.8 for MaSh and 0.2 for proximity. The probabilities are rough approximations based on experiments. Fig. 1 shows the mathematical curves; for example, the first suggestion given by MaSh is considered about 15 times more likely to appear in a successful proof than the 50th.

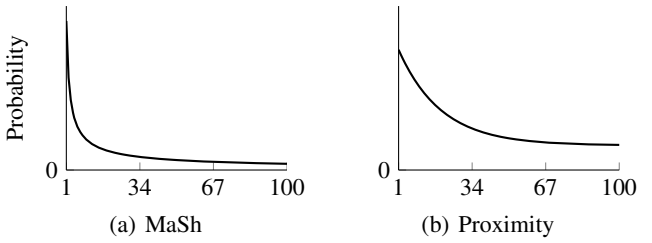


Fig. 1. Estimated probability of the jth fact's appearance in a proof

This notion of combining filters to define new filters is taken one step further by MeSh, a combination of *Me*Po and Ma*Sh*. Both filters are weighted 0.5, and both use the probability curve of Fig. 1(a).

Ideally, the curves and parameters that control the combination of filters would be learned mechanically rather than hard-coded. However, this would complicate and possibly slow down the infrastructure.

4.4 Automatic and Manual Control

All MaSh-related activities take place as a result of a Sledgehammer invocation. When Sledgehammer is launched, it checks whether any new facts, unknown to the visibility graph, are available. If so, it launches a thread to learn from their Isar proofs and update the graph. The first time, it may take about 10 s to learn all the facts in the background theories (assuming about 10 000 facts). Subsequent invocations are much faster.

If an automatic prover succeeds, the proof is immediately taught to MaSh (using MaSh.learn). The discharged (sub)goal may have been only one step in an unstructured proof, in which case it has no name. Sledgehammer invents a fresh, invisible name for it. Although this anonymous goal cannot be used to prove other goals, MaSh benefits from learning the connection between the formula's features and its proof.

For users who feel the need for more control, there is an `unlearn` command that resets MaSh's persistent state (using MaSh.unlearn); a `learn_isar` command that learns from the Isar proofs of all available facts; and a `learn_prover` command that invokes an automatic prover on all available facts, replacing the Isar proofs with successful ATP-generated proofs whenever possible.

4.5 Nonmonotonic Theory Changes

MaSh's model assumes that the set of facts and the visibility graph grow monotonically. One concern that arises when deploying machine learning—as opposed to evaluating its performance on static benchmarks—is that theories evolve nonmonotonically over time. It is left to the architecture around MaSh to recover from such changes. The following scenarios were considered:

- *A fact is deleted.* The fact is kept in MaSh's visibility graph but is silently ignored by Sledgehammer whenever it is suggested by MaSh.
- *A fact is renamed.* Sledgehammer perceives this as the deletion of a fact and the addition of another (identical) fact.
- *A theory is renamed.* Since theory names are encoded in fact names, renaming a theory amounts to renaming all its facts.
- *Two facts are reordered.* The visibility graph loses synchronization with reality. Sledgehammer may need to ignore a suggestion because it appears to be visible according to the graph.
- *A fact is introduced between two facts ϕ and ϕ'.* MaSh offers no facility to change the parent of ϕ', but this is not needed. By making the new fact a child of ϕ, it is considered during the computation of maximal vertices and hence visible.
- *The fact's formula is modified.* This occurs when users change the statement of a lemma, but also when they rename or relocate a symbol. MaSh is not informed of such changes and may lose some of its predictive power.

More elaborate schemes for tracking dependencies are possible. However, the benefits are unclear: Presumably, the learning performed on older theories is valuable and should be preserved, despite its inconsistencies. This is analogous to teams of humans developing a large formalization: Teammates should not forget everything they

know each time a colleague changes the capitalization of some basic theory name. And should users notice a performance degradation after a major refactoring, they can always invoke `unlearn` to restart from scratch.

5 Evaluations

This section attempts to answer the main questions that existing Sledgehammer users are likely to have: How do MaSh and MeSh compare with MePo? Is machine learning really helping? The answer takes the form of two separate evaluations.[3]

5.1 Evaluation on Large Formalizations

The first evaluation measures the filters' ability to re-prove the lemmas from three formalizations included in the Isabelle distribution and the *Archive of Formal Proofs* [19]:

Auth	Cryptographic protocols [26]	743 lemmas
Jinja	Java-like language [18]	733 lemmas
Probability	Measure and probability theory [15]	1311 lemmas

These formalizations are large enough to exercise learning and provide meaningful numbers, while not being so massive as to make experiments impractical. They are also representative of large classes of mathematical and computer science applications.

The evaluation is twofold. The first part computes how accurately the filters can predict the known Isar or ATP proofs on which MaSh's learning is based. The second part connects the filters to automatic provers and measures actual success rates.

The first part may seem artificial: After all, real users are interested in any proof that discharges the goal at hand, not a specific known proof. The predictive approach's greatest virtue is that it does not require invoking external provers; evaluating the impact of parameters is a matter of seconds instead of hours. MePo itself has been fine-tuned using similar techniques. For MaSh, the approach also helps ascertain whether it is learning the learning materials well, without noise from the provers. Two (slightly generalized) standard metrics, full recall and AUC, are useful in this context.

For a given goal, a fact filter (MePo, MaSh, or MeSh) ranks the available facts and selects the n best ranked facts $\Phi = \{\phi_1, \ldots, \phi_n\}$, with $\mathit{rank}(\phi_j) = j$ and $\mathit{rank}(\phi) = n+1$ for $\phi \notin \Phi$. The parameter n is fixed at 1024 in the experiments below.

The known proof Π serves as a reference point against which the selected facts and their ranks are judged. Ideally, the selected facts should include as many facts from the proof as possible, with as low ranks as possible.

Definition 1 (Full Recall). The *full recall* is the minimum number $m \in \{0, \ldots, n\}$ such that $\{\phi_1, \ldots, \phi_m\} \supseteq \Pi$, or $n+1$ if no such number exists.

Definition 2 (AUC). The *area under the receiver operating characteristic curve* (*AUC*) is given by

$$\frac{\left|\{(\phi, \phi') \in \Pi \times (\Phi - \Pi) \mid \mathit{rank}(\phi) < \mathit{rank}(\phi')\}\right|}{|\Pi| \cdot |\Phi - \Pi|}$$

[3] Our empirical data are available at `http://www21.in.tum.de/~blanchet/mash_data.tgz`.

		MePo		MaSh		MeSh	
		Full rec.	AUC	Full rec.	AUC	Full rec.	AUC
Isar proofs	*Auth*	430	79.2	190	93.1	**142**	**94.9**
	Jinja	472	73.1	307	90.3	**250**	**92.2**
	Probability	742	57.7	384	88.0	**336**	**89.2**
ATP proofs	*Auth*	119	93.5	198	92.0	**68**	**97.0**
	Jinja	163	90.4	241	90.6	**84**	**96.8**
	Probability	428	74.4	368	85.2	**221**	**91.6**

Fig. 2. Average full recall and average AUC (%) with Isar and ATP proofs

Full recall tells how many facts must be selected to ensure that all necessary facts are included—ideally as few as possible. The AUC focuses on the ranks: It gives the probability that, given a randomly drawn "good" fact (a fact from the proof) and a randomly drawn "bad" fact (a selected fact that does not appear in the proof), the good fact is ranked before the bad fact. AUC values closer to 1 (100%) are preferable.

For each of the three formalizations (*Auth*, *Jinja*, and *Probability*), the evaluation harness processes the lemmas according to a linearization (topological sorting) of the partial order induced by the theory graph and their location in the theory texts. Each lemma is seen as a goal for which facts must be selected. Previously proved lemmas, and the learning performed on their proofs, may be exploited—this includes lemmas from imported background theories. This setup is similar to the one used by Kaliszyk and Urban [17] for evaluating their Sledgehammer-like tool for HOL Light. It simulates a user who systematically develops a formalization from beginning to end, trying out Sledgehammer on each lemma before engaging in a manual proof.[4]

Fig. 2 shows the average full recall and AUC over all lemmas from the three formalizations. For each formalization, the statistics are available for both Isar and ATP proofs. In the latter case, Vampire was used as the ATP, and goals for which it failed to find a proof are simply ignored. Learning from ATP proofs improves the machine learning metrics, partly because they usually refer to fewer facts than Isar proofs.

There is a reversal of fortune between Isar and ATP proofs: MaSh dominates MePo for the former but performs slightly worse than MePo for the latter on two of the formalizations. The explanation is that the ATP proofs were found with MePo's help. Nonetheless, the combination filter MeSh scores better than MePo on all the benchmarks.

Next comes the "in vivo" part of the evaluation, with actual provers replacing machine learning metrics. For each goal from the formalizations, 13 problems were generated, with 16, 23 ($\approx 2^{4.5}$), 32, ..., 724 ($\approx 2^{9.5}$), and 1024 facts. Sledgehammer's translation is parameterized by many options, whose defaults vary from prover to prover and, because of time slicing, even from one prover invocation to another. As

[4] Earlier evaluations of Sledgehammer always operated on individual (sub)goals, guided by the notion that lemmas can be too difficult to be proved outright by automatic provers. However, lemmas appear to provide the right level of challenge for modern automation, and they tend to exhibit less redundancy than a sequence of similar subgoals.

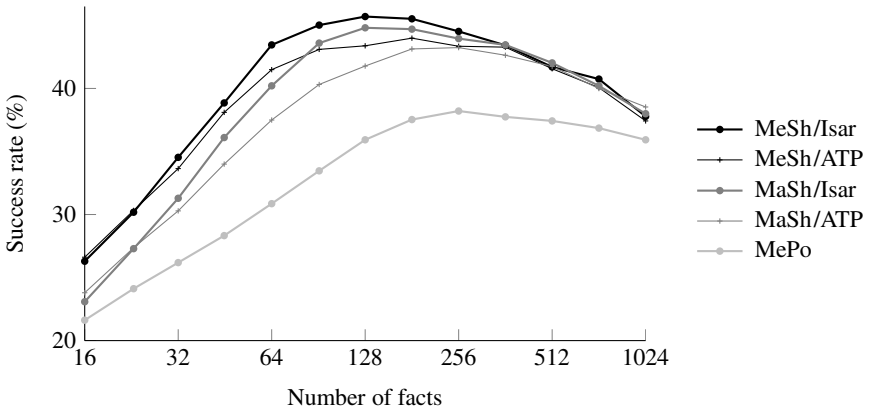


Fig. 3. Success rates for a combination of provers on *Auth* + *Jinja* + *Probability*

a reasonable uniform configuration for the experiments, types are encoded via the so-called polymorphic "featherweight" guard-based encoding (the most efficient complete scheme [8]), and λ-abstractions via λ-lifting (as opposed to the more explosive SK combinators).

Fig. 3 gives the success rates of a combination of three state-of-the-art automatic provers (Epar 1.6 [36], Vampire 2.6, and Z3 3.2) on these problems. Two versions of MaSh and MeSh are compared, with learning on Isar and ATP proofs. A problem is considered solved if it is solved within 10 s by any of them, using only one thread. The experiments were conducted on a 64-bit Linux server equipped with 12-core AMD Opteron 6174 processors running at 2.2 GHz. We observe the following:

- MaSh clearly outperforms MePo, especially in the range from 32 to 256 facts. For 91-fact problems, the gap between MaSh/Isar and MePo is 10 percentage points. (The curves have somewhat different shapes for the individual formalizations, but the general picture is the same.)
- MaSh's peak is both higher than MePo's (44.8% vs. 38.2%) and occurs for smaller problems (128 vs. 256 facts), reflecting the intuition that selecting fewer facts more carefully should increase the success rate.
- MeSh adds a few percentage points to MaSh. The effect is especially marked for the problems with fewer facts.
- Against expectations, learning from ATP proofs has a negative impact. A closer inspection of the raw data revealed that Vampire performs better with ATP (i.e., Vampire) proofs, whereas the other two provers prefer Isar proofs.

Another measure of MaSh and MeSh's power is the total number of goals solved for any number of facts. With MePo alone, 46.3% of the goals are solved; adding MaSh and MeSh increases this figure to 62.7%. Remarkably, for *Probability*—the most difficult formalization by any standard—the corresponding figures are 27.1% vs. 47.2%.

	MePo	MaSh	MeSh
E	55.0	49.8	**56.6**
SPASS	57.2	49.1	**57.7**
Vampire	55.3	49.7	**56.0**
Z3	53.0	51.8	**60.8**
Together	65.6	63.0	**69.8**

Fig. 4. Success rates (%) on Judgment Day goals

5.2 Judgment Day

The Judgment Day benchmarks [10] consist of 1268 interactive goals arising in seven Isabelle theories, covering among them areas as diverse as the fundamental theorem of algebra, the completeness of a Hoare logic, and Jinja's type soundness. The evaluation harness invokes Sledgehammer on each goal. The same hardware is used as in the original Judgment Day study [10]: 32-bit Linux servers with Intel Xeon processors running at 3.06 GHz. The time limit is 60 s for proof search, potentially followed by minimization and reconstruction in Isabelle. MaSh is trained on 9852 Isar proofs from the background libraries imported by the seven theories under evaluation.

The comparison comprises E 1.6, SPASS 3.8ds, Vampire 2.6, and Z3 3.2, which Sledgehammer employs by default. Each prover is invoked with its own options and problems, including prover-specific features (e.g., arithmetic for Z3; sorts for SPASS, Vampire, and Z3). Time slicing is enabled. For MeSh, some of the slices use MePo or MaSh directly to promote complementarity.

The results are summarized in Fig. 4. Again, MeSh performs very well: The overall 4.2 percentage point gain, from 65.6% to 69.8%, is highly significant. As noted in a similar study, "When analyzing enhancements to automatic provers, it is important to remember what difference a modest-looking gain of a few percentage points can make to users" [9, § 7]. Incidentally, the 65.6% score for MePo reveals progress in the underlying provers compared with the 63.6% figure from one year ago.

The other main observation is that MaSh underperforms, especially in the light of the evaluation of Sect. 5.1. There are many plausible explanations. First, Judgment Day consists of smaller theories relying on basic background theories, giving few opportunities for learning. Consider the theory *NS_Shared* (Needham–Shroeder shared-key protocol), which is part of both evaluations. In the first evaluation, the linear progress through all *Auth* theories means that the learning performed on other, independent protocols (certified email, four versions of Kerberos, and Needham–Shroeder public key) can be exploited. Second, the Sledgehammer setup has been tuned for Judgment Day and MePo over the years (in the hope that improvements on this representative benchmark suite would translate in improvements on users' theories), and conversely MePo's parameters are tuned for Judgment Day.

In future work, we want to investigate MaSh's lethargy on these benchmarks (and MeSh's remarkable performance given the circumstances). The evaluation of Sect. 5.1 suggests that there are more percentage points to be gained.

6 Related Work and Contributions

The main related work is already mentioned in the introduction. Bridges such as Sledgehammer for Isabelle/HOL, Miz$\mathbb{AR}$ [37] for Mizar, and HOL(y)Hammer [17] for HOL Light are opening large formal theories to methods that combine ATPs and artificial intelligence (AI) [20, 35] to help automate interactive proofs. Today such large theories are the main resource for combining semantic and statistical AI methods [39].[5]

The main contribution of this work has been to take the emerging machine-learning methods for fact selection and make them incremental, fast, and robust enough so that they run unnoticed on a single-user machine and respond well to common user-interaction scenarios. The advising services for Mizar and HOL Light [17, 32, 33, 37] (with the partial exception of MoMM [32]) run only as remote servers trained on the main central library, and their solution to changes in the library is to ignore them or relearn everything from scratch. Other novelties of this work include the use of more proof-related features in the learning (inspired by MePo), experiments combining MePo and MaSh, and the related learning of various parameters of the systems involved.

7 Conclusion

Relevance filtering is an important practical problem that arises with large-theory reasoning. Sledgehammer's MaSh filter brings the benefits of machine learning methods to Isabelle users: By decreasing the quantity and increasing the quality of facts passed to the automatic provers, it helps them find more, deeper proofs within the allotted time. The core machine learning functionality is implemented in a separate Python program that can be reused by other proof assistants.

Many areas are calling for more engineering and research; we mentioned a few already. Learning data could be shared on a server or supplied with the proof assistant. More advanced algorithms appear too slow for interactive use, but they could be optimized. Learning could be applied to control more aspects of Sledgehammer, such as the prover options or even MePo's parameters. Evaluations over the entire *Archive of Formal Proofs* might shed more light on MaSh's and MePo's strengths and weaknesses. Machine learning being a gift that keeps on giving, it would be fascinating to instrument a user's installation to monitor performance over several months.

Acknowledgment. Tobias Nipkow and Lawrence Paulson have, for years, encouraged us to investigate the integration of machine learning in Sledgehammer; their foresight has made this work possible. Gerwin Klein and Makarius Wenzel provided advice on technical and licensing issues. Tobias Nipkow, Lars Noschinski, Mark Summerfield, Dmitriy Traytel, and the anonymous reviewers suggested many improvements to earlier versions of this paper. The first author is supported by the Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO) project Learning2Reason. The second author is supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant Ni 491/14-1).

[5] It is hard to envisage all possible combinations, but with the recent progress in natural language processing, suitable ATP/AI methods could soon be applied to another major aspect of formalization: the translation from informal prose to formal specification.

References

1. The Mizar Mathematical Library, http://mizar.org/
2. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reasoning (April 2013) http://arxiv.org/abs/1108.3446
3. Alama, J., Kühlwein, D., Urban, J.: Automated and human proofs in general mathematics: An initial comparison. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 37–45. Springer, Heidelberg (2012)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Benzmüller, C., Paulson, L.C., Theiss, F., Fietzke, A.: LEO-II—A cooperative automatic theorem prover for higher-order logic (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 162–170. Springer, Heidelberg (2008)
6. Berghofer, S., Nipkow, T.: Proof terms for simply typed higher order logic. In: Aagaard, M., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 38–52. Springer, Heidelberg (2000)
7. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. J. Autom. Reasoning 51(1), 109–128 (2013), http://www21.in.tum.de/~blanchet/jar-smt.pdf
8. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 493–507. Springer, Heidelberg (2013)
9. Blanchette, J.C., Popescu, A., Wand, D., Weidenbach, C.: More SPASS with Isabelle—Superposition with hard sorts and configurable simplification. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 345–360. Springer, Heidelberg (2012)
10. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
11. Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer, Heidelberg (2012)
12. Carlson, A.J., Cumby, C.M., Rosen, J.L., Roth, D.: SNoW user guide. Tech. rep., C.S. Dept., University of Illinois at Urbana-Champaign (1999), http://cogcomp.cs.illinois.edu/papers/CCRR99.pdf
13. Dutertre, B., de Moura, L.: The Yices SMT solver (2006), http://yices.csl.sri.com/tool-paper.pdf
14. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 299–314. Springer, Heidelberg (2011)
15. Hölzl, J., Heller, A.: Three chapters of measure theory in Isabelle/HOL. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 135–151. Springer, Heidelberg (2011)
16. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Di Vito, B., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics, pp. 56–68. NASA Tech. Reports (2003)
17. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. CoRR abs/1211.7012 (2012), http://arxiv.org/abs/1211.7012
18. Klein, G., Nipkow, T.: Jinja is not Java. In: Klein, G., Nipkow, T., Paulson, L. (eds.) Archive of Formal Proofs (2005), http://afp.sf.net/entries/Jinja.shtml
19. Klein, G., Nipkow, T., Paulson, L. (eds.): Archive of Formal Proofs, http://afp.sf.net/
20. Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., Heskes, T.: Overview and evaluation of premise selection techniques for large theory mathematics. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 378–392. Springer, Heidelberg (2012)

21. Kühlwein, D., Urban, J.: Learning from multiple proofs: First experiments. In: Fontaine, P., Schmidt, R., Schulz, S. (eds.) PAAR-2012, pp. 82–94 (2012)
22. Matuszewski, R., Rudnicki, P.: Mizar: The first 30 years. Mechanized Mathematics and Its Applications 4(1), 3–24 (2005)
23. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
24. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
26. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. J. Comput. Secur. 6(1-2), 85–128 (1998)
27. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, A practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) IWIL-2010 (2010)
28. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. 15(2-3), 91–110 (2002)
29. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
30. Schulz, S.: First-order deduction for large knowledge bases. Presentation at Deduction at Scale 2011 Seminar, Ringberg Castle (2011), http://www.mpi-inf.mpg.de/departments/rg1/conferences/deduction10/slides/stephan-schulz.pdf
31. Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition—CASC-J6. AI Commun. 26(2), 211–223 (2013)
32. Urban, J.: MoMM—Fast interreduction and retrieval in large libraries of formalized mathematics. Int. J. AI Tools 15(1), 109–130 (2006)
33. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. J. Autom. Reasoning 37(1-2), 21–43 (2006)
34. Urban, J.: MaLARea: A metasystem for automated reasoning in large theories. In: Sutcliffe, G., Urban, J., Schulz, S. (eds.) ESARLT 2007. CEUR Workshop Proceedings, vol. 257. CEUR-WS.org (2007)
35. Urban, J.: An overview of methods for large-theory automated theorem proving. In: Höfner, P., McIver, A., Struth, G. (eds.) ATE-2011. CEUR Workshop Proceedings, vol. 760, pp. 3–8. CEUR-WS.org (2011)
36. Urban, J.: BliStr: The blind strategymaker. CoRR abs/1301.2683 (2013), http://arxiv.org/abs/1301.2683
37. Urban, J., Rudnicki, P., Sutcliffe, G.: ATP and presentation service for Mizar formalizations. J. Autom. Reasoning 50(2), 229–241 (2013)
38. Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, J.: MaLARea SG1— Machine learner for automated reasoning with semantic guidance. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 441–456. Springer, Heidelberg (2008)
39. Urban, J., Vyskočil, J.: Theorem proving in large formal mathematics as an emerging AI field. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS (LNAI), vol. 7788, pp. 240–257. Springer, Heidelberg (2013)
40. Wenzel, M.: Isabelle/Isar—A generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof—Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar, and Rhetoric, vol. 10(23). Uniwersytet w Białymstoku (2007)
41. Wenzel, M.: Parallel proof checking in Isabelle/Isar. In: Dos Reis, G., Théry, L. (eds.) PLMMS 2009, pp. 13–29. ACM Digital Library (2009)

Scalable LCF-Style Proof Translation

Cezary Kaliszyk[1] and Alexander Krauss[2]

[1] University of Innsbruck
cezary.kaliszyk@uibk.ac.at
[2] QAware GmbH
krauss@in.tum.de

Abstract. All existing translations between proof assistants have been notoriously sluggy, resource-demanding, and do not scale to large developments, which has lead to the general perception that the whole approach is probably not practical. We aim to show that the observed inefficiencies are not inherent, but merely a deficiency of the existing implementations. We do so by providing a new implementation of a theory import from HOL Light to Isabelle/HOL, which achieves decent performance and scalability mostly by avoiding the mistakes of the past. After some preprocessing, our tool can import large HOL Light developments faster than HOL Light processes them. Our main target and motivation is the Flyspeck development, which can be imported in a few hours on commodity hardware. We also provide mappings for most basic types present in the developments including lists, integers and real numbers. This papers outlines some design considerations and presents a few of our extensive measurements, which reveal interesting insights in the low-level structure of larger proof developments.

1 Introduction

The ability to exchange formal developments between different proof assistants has been on the wish list of users for as long as these systems have existed.

Most systems are designed in such a way that their proofs can, in principle, be exported into a form (for example tactics, or proof terms in type theory based systems, or kernel proof steps as it is in the case of HOL Light) that could be checked or imported by some other system. Implementations of such proof translations exist between various pairs of systems. However, they all have in common that they are very expensive in terms of both runtime and memory requirements, which makes their use impractical for anything but toy examples.

For instance, Obua and Skalberg [12] describe a translation from HOL Light and HOL4 to Isabelle/HOL. Their tool is capable of replaying the HOL Light standard library, but this takes several hours (on 2010 hardware). Similarly, Keller and Werner [10] import HOL Light proofs in Coq and report that the standard library requires ten hours to load in Coq. Note that the standard library takes less than two minutes to load in plain HOL Light, so there is roughly a factor 300 of overhead involved. Other descriptions of similar translations report comparable overhead [15,11].

So should we conclude that this sort of blowup is inherent in the approach and that proof translations must necessarily require lots of memory and patience? This paper aims

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 51–66, 2013.

to refute this common belief and show that the bad performance is merely a deficiency of the existing implementations. More specifically, we make the following contributions:

- We describe a new implementation of a proof translation from HOL Light into Isabelle/HOL, which performs much better than previous tools: After some preprocessing, the HOL Light standard library can be imported into Isabelle/HOL in 30 seconds, which reduces the overhead factor from 300 down to about 0.4.
- Large developments are routinely handled by our tool, such that we can import major parts of the formalized proof of the Kepler Conjecture, developed in the Flyspeck [3] project on normal hardware. Here, the overhead factor is a bit larger (with optimizations a factor of roughly 1.2).
- By providing better mappings of HOL Light concepts to Isabelle concepts, we obtain more natural results; in particular little effort is needed to map compatible but differently defined structures like integers and real numbers.
- We present various results obtained during our measurements, which yield some empirical insights about the low-level structure of larger formal developments.

Our work shares some visions with Hurd's OpenTheory project [6] but has a slightly different focus:

- Our translation is able to work directly on the sources of any HOL Light development, without requiring any modification, such as adding special proof recording commands. This is crucial when dealing with large developments, where such modifications would create significant work and versioning problems.
  In contrast, the OpenTheory setup still needs manual arrangements to the sources, which hinders its use out of the box. Therefore, it is also hard to assess its scalability to developments of Flyspeck's size.
- We do not focus on creating small, independent, reusable packages. Instead, our approach assumes a large development, which is treated as a whole. By default, all definitions and top-level theorems are converted, but filtering to a subset is easy.
- Our proof recording and replaying mechanisms are designed to be efficient for big proof developments, whereas OpenTheory focuses on maximum sharing. This means that our proof trace format has very short identifiers for the most common steps, and is designed to be easily parsed, whereas the OpenTheory is very explicit and includes special commands for storing objects of any kinds to be reused. The same proof traces written in the OpenTheory format would be roughly 10 times larger, which for proof traces whose sizes are measured in gigabytes becomes important.

This Paper is Structured as Follows. In Section 2, we give an overview of the architecture of our translation. In Section 3 we discuss memoization strategies that allow reducing the time and memory requirements of the translation processes. In Section 4 we present the statistics of inference steps involved in the translation and interesting statistics about HOL Light and Flyspeck discovered using our experiments and finally we conclude in Section 5 and present some outlook on the future work.

2 Architecture Overview

In this section we explain the four basic steps in our proof translation: the collection of theorems that need to be exported; the instrumented kernel used to export the inference steps performed in HOL Light; the offline proof processor, and the importer itself. The four components are presented schematically in Figure 1.

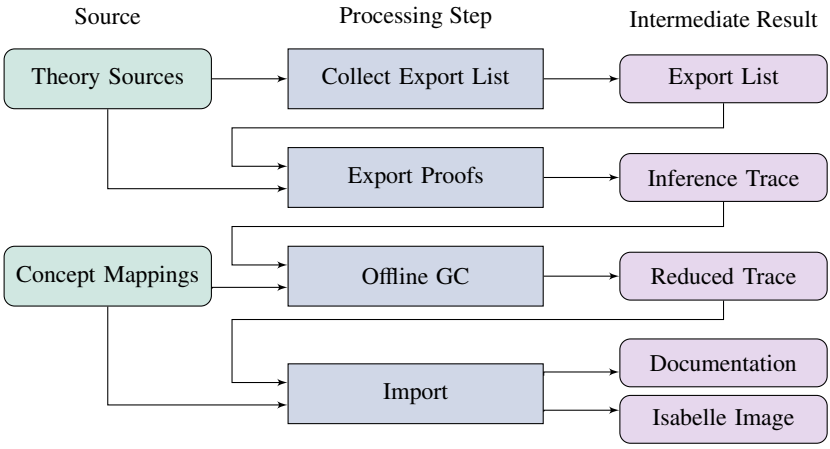


Fig. 1. The architecture of the translation mechanism

2.1 Collecting Theorems to Export

The first issue in implementing a proof translation mechanism is to choose which theorems are relevant and what are their canonical names. While many other proof assistants manage a list of named lemmas explicitly in some form of table, HOL Light simply represents lemmas as OCaml values of a certain type, bound on the interactive toplevel.

We first tried to follow a strategy similar to Obua and Skalberg [12], where statements to export are collected heuristically, by detecting certain idioms in the theory sources, such as the use of the function prove in the context of a toplevel binding. However, this rather superficial analysis is quite incomplete and fails to detect non-standard means of producing lemmas occur frequently in larger developments. It also produces false positives for bindings local to a function or module.

Instead of such guesses based on surface syntax, it is more practical to rely on the existing update_database [5] functionality, which can produce a list of name-theorem pairs accessible from the toplevel by analyzing OCaml's internal data structures. The result can be saved to a file and loaded into a table that maps statements to names, before starting the actual export. Whenever a new theorem object is created, we can then look up the corresponding name, if any, in the table. The benefit of this mechanism is that the original theory sources do not need to be modified.

2.2 Exporting the Inference Trace

To export the performed proofs, we use a patched version of the HOL Light kernel, which is modified to record all inference steps. Our recorded HOL Light session is a sequence of steps according to the grammar in Fig. 2. Each step constructs a new type, term or theorem, and the new object is implicitly assigned an integer id from an increasing sequence (separately for each kind of object), by which it can be referenced later. The arguments of proof steps are either identifiers (such as names of variables or constants), or references to previous proof steps, with an optional tag denoting the deletion of the referenced object.

⟨Step⟩ ::=		A step is either:
	⟨Typ⟩	a type construction step
	| ⟨Trm⟩	a term construction step
	| ⟨Thm⟩	a theorem construction step
⟨Typ⟩ ::=		A type step is either:
	TyVar ⟨id⟩	a named type variable
	| TyApp ⟨id⟩ ⟨ref⟩*	a type application with a list of subtypes
⟨Trm⟩ ::=		A term step is either:
	Var ⟨id⟩ ⟨ref⟩	a named variable
	| Const ⟨id⟩ ⟨ref⟩	a named constant with a type, the type is necessary for polymorphic constants
	| App ⟨ref⟩ ⟨ref⟩	an application
	| Abs ⟨ref⟩ ⟨ref⟩	an abstraction; in valid HOL Light abstractions the first subterm is always a term that is a variable
⟨Thm⟩ ::=		A term step is either:
	| Refl ⟨ref⟩	Reflexivity of a term
	| Trans ⟨ref⟩ ⟨ref⟩	Transitivity of two theorems
	| Comb ⟨ref⟩ ⟨ref⟩	Application of two theorems
	| Abs ⟨ref⟩ ⟨ref⟩	Abstraction of a term and a theorem
	| Beta ⟨ref⟩	β-reduction of a term
	| Assum ⟨ref⟩	Assumption of a term
	| Eqmp ⟨ref⟩ ⟨ref⟩	Equality *modus-ponens* of two theorems
	| Deduct ⟨ref⟩ ⟨ref⟩	Anti-symmetric deduction of two theorems
	| Inst ⟨ref⟩ ⟨ref⟩*	Type substitution with a theorem and a list of types
	| Subst ⟨ref⟩ ⟨ref⟩*	Term substitution with a theorem and a list of terms
	| Axiom ⟨ref⟩	Axiom with a term
	| Defn ⟨id⟩ ⟨ref⟩	Definition of an identifier to a term
	| TyDef ⟨id⟩ ⟨id⟩ ⟨id⟩ ⟨ref⟩ ⟨ref⟩ ⟨ref⟩	Type definition with three identifiers, two terms and a proof of existence
	| Save ⟨id⟩ ⟨ref⟩	Assign a name to a theorem

Fig. 2. The grammar of our export format. Identifiers ⟨id⟩ are strings with the same characters allowed as in the input system and references ⟨ref⟩ are integers

The keywords in the grammar are represented as single characters in our concrete trace format. The trace is generated on the fly and piped to a *gzip* process, which compresses it and writes it to disk.

HOL Light defines the type of theorems to be an abstract type and the types of terms and types to be private types. For the abstract type thm, we can achieve the numbering simply by adding an integer tag, which is filled from a global proof step counter when the theorem is constructed.

Hence, the proof steps are recorded in the order in which they occur. This instrumentation is local to the kernel (module `fusion.ml`), which encapsulates all theorem construction and destruction operations. No changes are required in other files.

For terms and types this is not so easy, as in OCaml for objects of private types matching is possible. This means that we cannot add tags without breaking client code. However, just writing out terms naively would destroy all sharing and lead to significant blowup. To make the process manageable, we need to preserve at least some of the sharing present in memory. Thus we employ memoization techniques that partly recover the sharing of types and terms. Section 3 describes these techniques; all of them produce traces in the format described above.

There is also a choice of the basic proof steps that get written. We have decided to export the minimal set of inference rules that covers HOL Light. This is in opposition to Obua and Skalberg's export where some of the proof rules get exported as multiple steps (Deduct gets exported as three rules that are easier to import) whereas other rules get optimized proofs. We found out that such *ad-hoc* optimizations yield only very small improvements in efficiency (below 2%), but unnecessarily complicate the code and make it more prone to errors.

2.3 Offline Garbage Collection

While our trace records the creation of all relevant objects, it does not contain information about their deletion. However, this information is equally important if we do not want to create a memory leak in the Isabelle import, which would have to keep all objects, even though they have long been garbage-collected in the HOL Light process.

While the information about object deletion is not directly present in the trace, it can easily be recovered by marking the last reference to each object specially as an indication that the importer should drop the reference to the object after using it.

We do this in a separate processing step, which in addition performs an offline garbage collection on the whole graph and throws away all objects that do not contribute to the proofs of a named theorem.

The offline processor takes two inputs: the inference trace and a list of theorems that should be replaced by Isabelle theorems during import. The program first removes all the steps that are not needed when importing. This includes theorems that were created but never used (for example by proof search procedures that operate on theorems), theorems that are mapped to their Isabelle counterparts, and their transitive dependencies. Next, the last occurrences of a reference to any type, term or theorem is marked, so that the importer can (after using the object as a dependency) immediately forget it. Currently, our offline processor reads the whole dependency graph into memory. This is feasible even for developments of the size of Flyspeck. Hypothetical, for larger developments

this could be replaced by an algorithm that does several passes on the input and requires less memory.

The output of this step has the same format as the raw trace. We call it the *reduced trace*.

2.4 Import

The actual import into Isabelle now amounts to replaying the reduced trace generated in the previous step. In addition, the import is configured with the mappings of types, constants and theorems to the respective Isabelle concepts.

Replaying the trace is conceptually straightforward: we simply replay step by step, keeping integer-indexed maps to look up the required objects needed for each proof step.

We use Isabelle's `cterm` type (an abstract type representing type-checked terms) to store terms, and similarly for types. This avoids repeated type-checking of terms and reduces import runtime by a factor of two, with a slight increase in memory use.

We do not attempt to generate theory source text, which was a major bottleneck and source of problems in Obua and Skalberg's approach, since re-parsing the generated theories is time-consuming, and the additional layer of build artifacts only makes the setup unnecessarily complex with little benefit. The only advantage of the generated theories were that users could use them to inspect the imported material. For this purpose, we instead generate a documentation file (in LaTeX and HTML), which lists lemma names and statements. Excerpts from the documentation can be seen in Figures 3 and 4. We also provide the complete rendered documentation for the Flyspeck import at http://cl-informatik.uibk.ac.at/~cek/import/.

```
thm RIGHT_OR_DISTRIB:
  ∀(p::bool) (q::bool) r::bool. ((p ∨ q) ∧ r) = (p ∧ r ∨ q ∧ r)
thm FORALL_SIMP:
  ∀t::bool. (∀x::'a. t) = t
thm EXISTS_SIMP:
  ∀t::bool. (∃x::'a. t) = t
thm EQ_CLAUSES:
  ∀t::bool. (True = t) = t ∧ (t = True) = t ∧ (False = t) = (¬ t) ∧ (t = False) = (¬ t)
```

Fig. 3. Cropped printout of the HTML documentation of the imported HOL Light library

After replaying the trace in Isabelle, it can be used interactively or saved to an image (using Isabelle's standard mechanisms) for later use.

Concept Mappings. In general it is desirable to map HOL Light concepts to existing Isabelle concepts whenever they exist instead of redefining them during the import. This makes it easier to use the imported results and combine them with existing ones. This is a form of *theory interpretation*.

thm opposite_hypermap_plain:

$\forall H.\ plain\_hypermap\ H \longrightarrow plain\_hypermap\ (opposite\_hypermap\ H)$

thm opposite_components:

$\forall H\ x.\ dart\ (opposite\_hypermap\ H) = dart\ H \wedge node\ (opposite\_hypermap\ H)\ x = node\ H\ x \wedge face\ (opposite\_hypermap\ H)\ x = face\ H\ x$

thm opposite_hypermap_simple:

$\forall H.\ simple\_hypermap\ H \longrightarrow simple\_hypermap\ (opposite\_hypermap\ H)$

thm hypermap_eq_lemma:

$\forall H.\ tuple\_hypermap\ H = (dart\ H,\ edge\_map\ H,\ node\_map\ H,\ face\_map\ H)$

Fig. 4. Cropped printout of the LaTeX generated PDF documentation of Flyspeck imported to Isabelle. The documentation can be generated with or without type annotations, and we chose to present the type annotations in Fig. 3 and no types here.

There are two scenarios: First, if the definition of the constant or type (typedef) from the original system can be derived in the target system, it is enough to replace the definition with the derived theorem. Second, if a definition or type is not defined in the same way and the original definition cannot be derived, the procedure is more involved. This is the common case for more complex definitions.

Consider the function HD which returns the first element of a given non-empty list. Its result on the empty list is some arbitrary unknown value, but while HOL Light makes use of the Hilbert operator ϵ, Isabelle/HOL uses list_rec undefined, which is an artifact of the tools used. Both these terms represent "arbitrary" values, but they are not provably equal. However, since there are no theorems in HOL Light that talk about the head of an empty list, we can get away with it.

In general, we can replace any set of characteristic properties from which all translated results are derived and which is provable in the target system. This need not be the actual definition of the constant. This also means that some lemmas that are merely used to derive the characteristic properties will not be translated. This requires some dependency analysis which is hard to do during the actual export or import, which merely write and read a stream. An offline processor (our proof garbage collector) can verify that all needed theorems are mapped: if a constant or type is mapped, the theorems that define this type or constant need to be mapped as well. Given that the definition cannot be mapped, its direct dependencies need to be mapped. This means, that adding type and constant maps can be done completely offline: as we prove mapping theorems manually and add them to the list of maps, the offline processor either confirms that the import will succeed or gives a list of missing theorems.

Obua and Skalberg's import attempted to resolve mappings during import, which would make mapping non-trivial concepts like the real numbers a tedious trial-and-error experience.

Mapping of constants can be provided simply by giving a theorem attribute in Isabelle, for example to map the HOL Light constant FST to the Isabelle/HOL constant fst it is enough to prove the following:

```
lemma [import_const FST]:
  "fst = (λp::'A × 'B. SOME x::'A. ∃y::'B. p = (x, y))"
  by auto
```

and similarly to map a type:

```
lemma [import_type prod ABS_prod REP_prod]:
  "type_definition Rep_prod Abs_prod (Collect
    (λx::'A ⇒ 'B ⇒ bool. ∃a b. x = Pair_Rep a b))"
  using type_definition_prod[unfolded Product_Type.prod_def] by simp
```

There is one more issue that we need to address. Even if the the natural numbers of HOL Light are mapped to the natural numbers of Isabelle, the binary representation of nat in Isabelle is different from that of num in HOL Light. To make them identical, a set of rewrite rules is applied that rewrites the constant NUMERAL applied to bits, to the Isabelle version thereof.

3 Time and Memory Comparison of Flyspeck vs Import

In this Section we discuss the memoization techniques used to reduce the import time and memory footprint of the processing steps as well as the import time and give some comparisons of the processes involved in Import with the original ones of HOL Light.

As we have noticed in Section 2 when writing the trace the sharing between terms (and types) is lost. The number and size of types in a typical HOL Light development is insignificant in comparison with the number and size of terms; which is why we will focus on optimizing the terms; however the same principles are used for types.

There are many ways in which the problem can be addressed; the simplest is to simply write out all the terms. This is equivalent to not doing any sharing; and came out to be infeasible in practice. Even when writing a compressed trace after 3 weeks runtime, the compressed trace was 500GB, without having got past HOL Light's Multivariate library (a prerequisite of Flyspeck).

So is sharing always good? Then the ultimate goal would be to achieve complete sharing, where the export process keeps all the terms that have been written so far, and whenever a new term is to be written it is checked against the present ones. However, keeping all terms in memory obviously does not scale, as the memory requirements would grow linearly with the size of the development. (In fact, this was true in Obua and Skalberg's implementation, where recorded proofs were kept in-memory eternally.) Moreover, the import stage becomes equally wasteful, since terms are held in memory between any two uses, where it would be much cheaper to rebuild them when needed. Running Flyspeck with this strategy requires 70GB RAM for the export and 120GB for the import stage.

Instead, we employ a least-recently-used strategy, which keeps only a fixed number of N entries in a cache and discards those that were not referenced for the longest

time. While this does not necessarily reflect the actual life cycle of terms in the original process, it seems to be a good enough approximation to be useful. In particular, it avoids wasting memory by keeping unused objects around for a long time. Our data structures are built on top of OCaml's standard maps in a straightforward manner, and all operations are $O(\log N)$.

We show the impact of the size of the term cache on the export time and memory footprint in Figure 5. The graph shows two datasets: the core of HOL Light and the VOLUME_OF_CLOSED_TETRAHEDRON lemma from Flyspeck. We chose the former, as it represents a typical HOL Light development, consisting of a big number of regular size lemmas, and we chose the latter, as it creates a big proof trace. The big trace is created using the REAL_ARITH decision procedure which implements the Gröbner bases procedure [4].

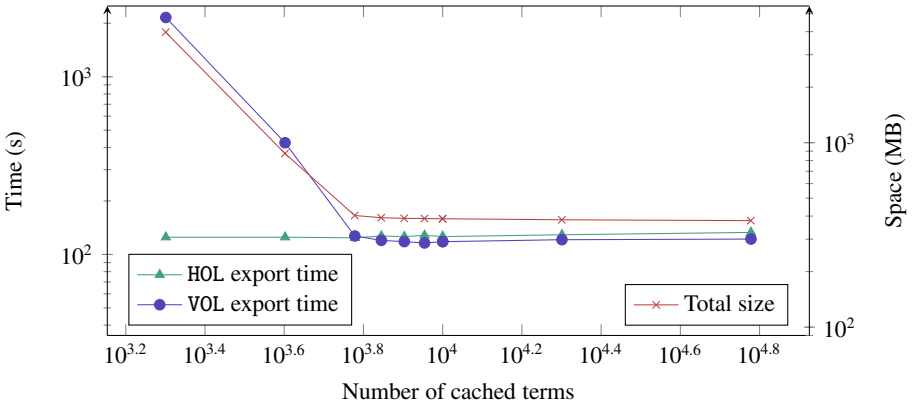


Fig. 5. Time required to write the proof trace and the size of the resulting trace as a function of the term cache size. We compare the time of the HOL Light standard library (HOL) and the VOLUME_OF_CLOSED_TETRAHEDRON theorem (VOL).

As we can see the size of the term cache has very little impact on the time and size of HOL Light standard library. However for a proof which constructs a big number of bigger terms using a very small cache increases the export time and trace size exponentially. This means that the size needs to be adjusted to the size of the terms produced by the decision procedures and if this is not known a biggest possible term cache size is advisable.

We performed a similar experiment for caching proofs. The first idea is to reuse the proof number if a theorem with same statement has already been derived. This can however cause problems, given that there are some mappings between constants or types defined in different ways. For example the theorems that lead to the definition of such a type cannot be directly reused from the cache. This means that unnamed theorems used before a mapping cannot be reused after the mapping. To overcome this difficulty we chose to clear the theorem cache at every named theorem. This may lose a small amount of sharing but prevents issues with adding constant and type maps. We have computed the impact that the size of the proof cache has on the resulting trace and the export time in Figure 6.

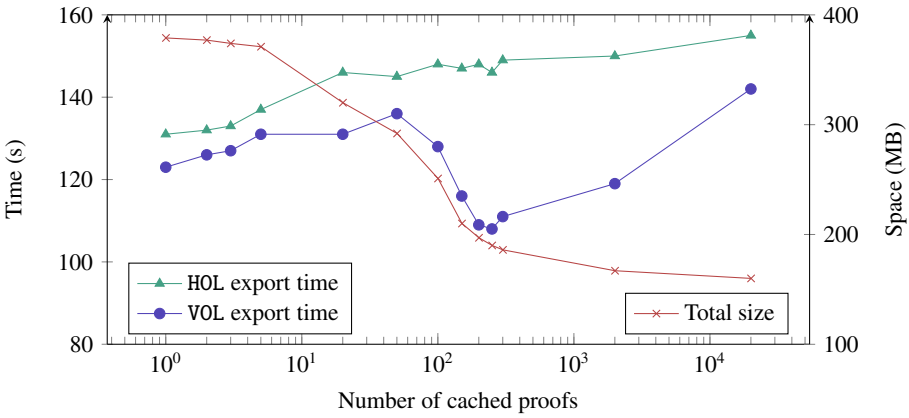


Fig. 6. Time required to write the proof trace and the size of the resulting trace as a function of the proof cache size. We compare the time of the HOL Light standard library (`HOL`) and the VOLUME_OF_CLOSED_TETRAHEDRON theorem (`VOL`).

As we can see for typical proof steps (HOL Light standard library) with more sharing the export time increases however not significantly. This is not the case for a proof which constructs big intermediate results. In such a case there is an optimal number of proofs to cache and with a bigger number the complexity of comparing the proofs with the cache increases the time. Since in a typical HOL Light development the optima for the different proofs may differ, we instead choose to use a small proof cache. In Section 4 we will see that the decrease in the space does not lead to a significant decrease in the Import time. We have compared the memory usage of the OCaml process writing the trace with the memory of the PolyML process doing the Import (Fig. 7 and the two are roughly comparable, with two exceptions. PolyML is much less conservative when allocating memory, and quickly uses all available memory, however most of it is reclaimed by major collections. Also due to the garbage collection running in a separate thread in PolyML major collections happen more often than in OCaml.

4 Statistics Over Flyspeck

In this section we look at various statistics that can be discovered when analyzing the proof-recorded Flyspeck.

4.1 Dependencies and Steps Statistics

We first look at the dependencies between theorems, terms and types. The following table shows the total number of theorem steps, term steps and type steps in the proof trace. The table includes four rows, first two are for full sharing, second two are for term caching.

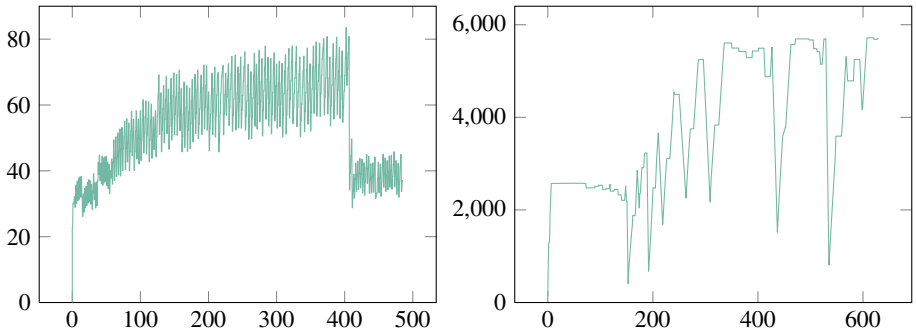
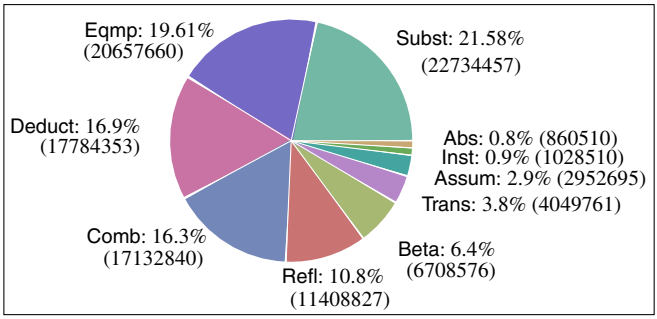


Fig. 7. Comparing the export memory (left) with import memory (right) for the core HOL Light together with the VOLUME_OF_CLOSED_TETRAHEDRON theorem. Export memory is computed with OCaml garbage collection statistics in millions of live words. Import memory in bytes.

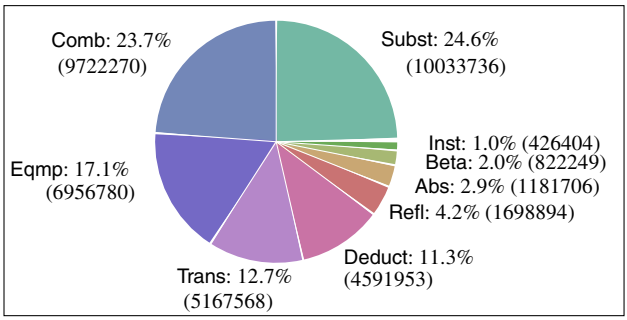
Strategy	Types	Terms	Theorems
Full sharing	173,800	51,448,207	146,120,269
Full sharing processed	130,936	13,066,288	40,802,070
Term caching	173,800	101,846,215	420,253,109
Term caching processed	146,710	23,318,639	194,541,803

We first notice that the number of types is insignificant relative to the number of terms or theorems, and the number of types that can be removed by processing is only 16–25%. For terms, the number of terms with full sharing is 50% of that with caching, which means that the overhead is still quite big; and the overhead reduces only to 45% with the offline GC. Sharing has the biggest effect on theorems: the shared theorems are 34% of the cached ones. Finally, offline GC lets reduce the number of terms and theorems roughly a quarter of the recorded ones, which is a huge improvement.

Next we will look at the exact inference steps that were derived but not needed. We have computed the numbers of inference steps of each kind and their percentages for the steps that were derived but the offline processor could remove them:

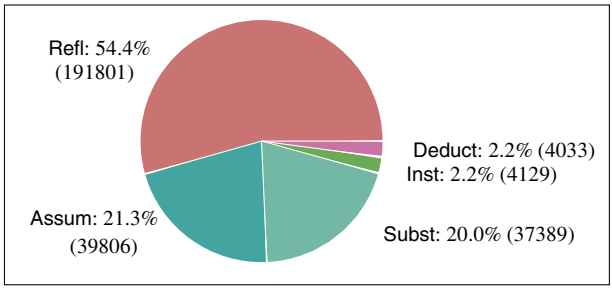


We compare the above to the needed steps (the steps left by the garbage collection):



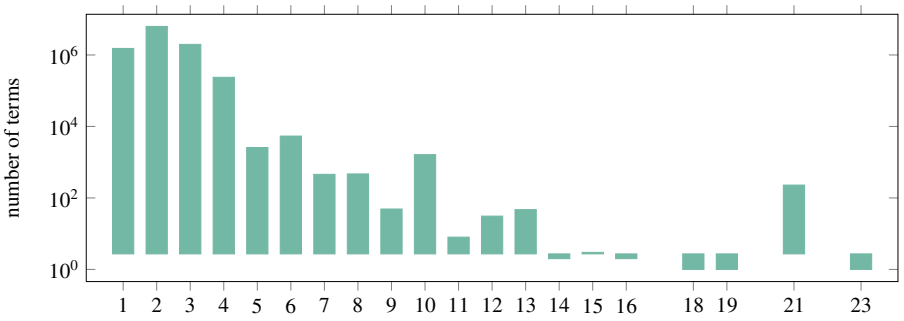
In both cases the methods that do complicated computation (Subst, Eqmp, Comb) dominate; however for the unneeded diagram the very cheap methods Refl and Assum occur much more often.

We next compare it to the steps that were derived multiple times. 2,515,531 theorem steps have been derived multiple times with exactly same dependencies. It is quite interesting, that many of these steps are the same ones repeated over and over again. This suggests that the steps are performed repeatedly by decision procedures or even by the simplifier. The unique repeated steps are only 187,164 which is just 7% of the repeated steps! We have analyzed the kinds of steps that have been derived multiple times:



Here we see a dominance of Refl and Assum which are the two cheapest steps. However the third steps — Subst — is not a cheap one; and this is where the main advantage of caching or sharing comes in.

We have also looked at the complexity of individual Subst steps; even if the maximum is 23, the average number of term pairs in a substitution is 2.098, and the distribution is as follows:



We see that there are 229 substitutions with 21 simultaneously substituted terms; however the time consumed by these substitutions is caused by the size of the terms involved.

4.2 Theorem Name Statistics

There are 622 theorems that have been assigned more than one name. In certain cases useful theorems from another module have been given a different name in a later one; however in certain cases theorems have been rederived, sometimes in completely different ways. We have computed the statistics:

Unique statements	Number of names	Canonical name
1	16656	
2	567	
3	46	
4	5	
5	2	Trigonometry1.ups_x, Sphere.aff
7	1	Trigonometry.UNKNOWN
11	1	REAL_LE_POW

In certain cases, theorems with meaningful names have been assigned also random names; but there are a few cases where meaningful names have been assigned to theorems twice. There are multiple possible reasons for this and we present here a few common cases, from the more trivial ones to more involved ones.

- REAL_LT_NEG2 and REAL_LT_NEG both derived using the real decision procedure in the same file just 70 lines further.
- Topology.LEMMA_INJ and Hypermap.LEMMA_INJ the proof is an exact copy, still both are processed.
- REAL_ABS_TRIANGLE and Real_ext.ABS_TRIANGLE first one derived using a decision procedure, second one with a complete proof.

- INT_NEG_NEG and INT_NEGNEG both derived using the quotient package from real number theorems called differently.
- CONVEX_CONNECTED_1_GEN and CONNECTED_CONVEX_1_GEN where one is derived from the other.

4.3 Translation Time and Size Statistics

Given the best cache sizes computed in Section 3 and the statistics over the steps we have computed the times and sizes of the various stages of Import evaluated on the whole Flyspeck development:

Phase	Time	Size
Collect Export List	4h	18MB
Export Proofs	10h	3692MB
Offline GC	48m	1089MB
Import without optimizations	54h	

The created Isabelle image can be loaded in a few seconds and memory allocated by the underlying PolyML system is 2.7GB which is almost same as the 2.7GB allocated by HOL Light. The time required to create the image is by a factor of 13 longer than the time required to run Flyspeck. This is still quite a lot, and is possible to do it once, but not if such a development must be translated routinely. As we have discovered in Section 4.1, this is caused by decision procedures; in fact REAL_ARITH (an implementation of Gröbner bases) is what creates the huge terms and substitutions that constitute a big part of the proof trace and import time.

To further optimize the Import time we tried to map the two most expensive calls to this decision procedure (using the theorem mapping mechanism) to two counterpart theorems proved using a similar decision procedure in Isabelle. The algebra tactic [2] can solve the goals in Isabelle in a similar time as that needed in HOL Light:

Phase	Time	Size
Offline GC	48m	964MB
Import with optimizations	4.5h	

The reduced proof trace is 11% smaller, but the import time becomes roughly equal to processing of the theories with HOL Light.

5 Conclusion

We have presented a new implementation of a theory import from HOL Light to Isabelle/HOL, designed to achieve decent performance. The translation allows mapping the concepts to their Isabelle counterparts obtaining natural results. By analyzing the proof trace of Flyspeck we have also presented a number of statistics about a low-level structure of a big formal development.

The code of our import mechanism has been included in Isabelle, together with a component for loading the core HOL Light automatically and the documentation for it.

The formalization includes the mappings of all the basic types present in both developments including types that are not defined in the same way, such as lists, integers and real numbers. In total, 97 constructors have been mapped with little effort. It is easy to generate similar components for other HOL Light developments. The code is 5 times smaller than the code of Obua and Skalberg's Import. The trace recording component for HOL Light can be obtained from http://cl-informatik.uibk.ac.at/~cek/import/.

In our development we defined a new format for proof exchange traces, despite the existence of other exchange formats. We have tried writing the proof trace in the OpenTheory format, and it was roughly 10 times bigger. For the proof traces whose sizes are measured in gigabytes such an optimization does make sense; however the is it conceivable to share the Import code with other formats.

The processed trace generated by this work is already used by machine learning tools for HOL Light to provide proof advice [7,8].

5.1 Future Work

The most obvious future work is testing our export on other HOL Light developments, including the most interesting ones which are not formalized in Isabelle, for example the developments from Wiedijk's 100 theorems list [14] and Hilbert Axiom Geometry. Similarly the work can be extended to work with different pairs of provers, in particular non-HOL-based ones, or the Common HOL Platform [1] intended as a minimal base for sharing HOL proofs.

A different line of work could be to automate the mapping of results of decision procedures. We have tried to export the steps performed by REAL_RING, REAL_ARITH, or REAL_FIELD as a single proof step (modifying not only the kernel, but also the theories in which these procedures are defined). Unfortunately for each of these, there exists at least one goal that it solved, but which could not be solved by algebra; this needs to be investigated further.

The statistics performed on the repository allow for an easy discovery of duplicate proofs and multiple names given to same theorems. This can be used to streamline the original developments. Conversely, for imported libraries that match ones present in the target system, analyzing the theorems that are not present may lead to the discovery of interesting intermediate lemmas.

Acknowledgements. Makarius Wenzel originally sparked our interest in the topic by claiming that efficient translation should be possible, even though none existed yet. Johannes Weigend deserves thanks for his *Software EKG* toolset and methodology [13], which was a tremendous help in analyzing the various datasets from our measurements.

References

1. Adams, M.: Introducing HOL Zero - (extended abstract). In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 142–143. Springer, Heidelberg (2010)

2. Chaieb, A., Nipkow, T.: Proof synthesis and reflection for linear arithmetic. J. Autom. Reasoning 41(1), 33–59 (2008)
3. Hales, T.C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., Zumkeller, R.: A revision of the proof of the Kepler conjecture. Discrete & Computational Geometry 44(1), 1–34 (2010)
4. Harrison, J.: Automating elementary number-theoretic proofs using Gröbner bases. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 51–66. Springer, Heidelberg (2007)
5. Harrison, J., Zumkeller, R.: update_database module. Part of the HOLLight distribution
6. Hurd, J.: The OpenTheory standard theory library. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 177–191. Springer, Heidelberg (2011)
7. Kaliszyk, C., Urban, J.: Initial experiments with external provers and premise selection on HOL Light corpora. In: Fontaine, P., Schmidt, R., Schulz, S. (eds.) PAAR (to appear 2012)
8. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. CoRR, abs/1211.7012 (2012)
9. Kaufmann, M., Paulson, L.C. (eds.): ITP 2010. LNCS, vol. 6172. Springer, Heidelberg (2010)
10. Keller, C., Werner, B.: Importing HOL Light into Coq. In: Kaufmann and Paulson [9], pp. 307–322
11. Krauss, A., Schropp, A.: A mechanized translation from higher-order logic to set theory. In: Kaufmann and Paulson [9], pp. 323–338
12. Obua, S., Skalberg, S.: Importing HOL into Isabelle/HOL. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 298–302. Springer, Heidelberg (2006)
13. Weigend, J., Siedersleben, J., Adersberger, J.: Dynamische Analyse mit dem Software-EKG. Informatik Spektrum 34(5), 484–495 (2011)
14. Freek Wiedijk. Formalizing 100 theorems, `http://www.cs.ru.nl/ freek/100/`
15. Wong, W.: Recording and checking HOL proofs. In: Schubert, E.T., Windley, P.J., Alves-Foss, J. (eds.) HUG 1995. LNCS, vol. 971, pp. 353–368. Springer, Heidelberg (1995)

Lightweight Proof by Reflection Using a Posteriori Simulation of Effectful Computation

Guillaume Claret[1], Lourdes del Carmen González Huesca[1], Yann Régis-Gianas[1], and Beta Ziliani[2]

[1] PPS, Team πr^2 (University Paris Diderot, CNRS, and INRIA)
{guillaume.claret,lgonzale,yann.regis-gianas}@pps.univ-paris-diderot.fr
[2] Max Planck Institute for Software Systems (MPI-SWS)
beta@mpi-sws.org

Abstract. Proof-by-reflection is a well-established technique that employs decision procedures to reduce the size of proof-terms. Currently, decision procedures can be written either in Type Theory—in a purely functional way that also ensures termination— or in an effectful programming language, where they are used as oracles for the certified checker. The first option offers strong correctness guarantees, while the second one permits more efficient implementations.

We propose a novel technique for proof-by-reflection that marries, in Type Theory, an effectful language with (partial) proofs of correctness. The key to our approach is to use *simulable* monads, where a monad is simulable if, for all terminating reduction sequences in its equivalent effectful computational model, there exists a witness from which the same reduction may be simulated *a posteriori* by the monad. We encode several examples using simulable monads and demonstrate the advantages of the technique over previous approaches.

1 Introduction

In Type Theory, types may embed computation, thereby allowing for a proof technique called *proof by reflection*. This technique reduces the time to typecheck a proof by replacing potentially large proof-terms by small proof-terms, whose verification consists of computing at the type level.

For instance, say that verifying a proof Δ of $\mathsf{P\ a}$ is computationally expensive, for $\mathsf{P : A \to Prop}$, with A a type, and $\mathsf{a : A}$. Let B be a type such that there exists an *interpretation* function I from B to A, and a decision procedure $\mathsf{D : B \to bool}$. Furthermore, let us assume that D *decides* P, that is, there is a theorem

$$\mathsf{sound\ :\ \forall x : B, D\,x = true \to P\,(I\,x)}$$

which states that for every element x of B, if the decision procedure returns true for this element, then property P holds for the interpretation of x. With these definitions at hand, then if we have some $\mathsf{b : B}$ such that $\mathsf{I\,b = a}$, we can replace the original proof-term Δ with

$$\mathsf{sound\ b\ (refl\_equal\ true)}$$

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 67–83, 2013.

where refl_equal has type $\forall x : \mathsf{bool}, x = x$. Typechecking that the proof-term above has the expected type (P a) effectively amounts to (i) executing the procedure D on b, (ii) checking that its result is equal to true, (iii) and checking that the interpretation of b is equal to a.[1]

Previous works [11,4] have exposed several advantages and weaknesses of proof by reflection, especially in comparison with the traditional LCF proof style [9]. In a nutshell, the former is considered more robust to change, while the latter is easier to write. Indeed, proving by reflection has a price: the decision procedure D must usually be written in a constrained programming language with only total functions and no imperative features. Furthermore, soundness proofs are often complex and thus difficult to construct [8]. These two problems, the restricted language and the need for keeping the proof of soundness simple, incite the proof developer to write inefficient decision procedures, which is regrettable since proof search is intrinsically a computationally expansive process.

There is a variation of proof by reflection that alleviates some of these problems, called *certifying* proof by reflection [3,10]. In this technique, the decision procedure is written in a general purpose programming language, and used by the proof assistant as an *untrusted oracle*. The decision procedure returns a certificate, which is mechanically verified by the proof assistant *via* a certificate checker written in Type Theory. This checker and its proof of correctness are usually kept simple, whereas the untrusted oracle can be as sophisticated as necessary to implement the decision procedure efficiently. However, this technique has its drawbacks. First, it is not as efficient as one may expect, as the certificate embedded in the resulting proof-term can be large and, in addition, there is a cost of executing the oracle, plus verifying the certificate with the checker. Second, the proof developer is forced to write the certificate checker *and* the decision procedure (or adapt an existing one in order to produce the certificate). Third, the implementation of an oracle usually gives only weak guarantees about its applicability (a perfectly valid but useless oracle could fail on every input) because proving completeness properties about a program written in a general purpose programming language is notoriously hard.

In this paper, we propose a novel style of proof by reflection that allows for writing an *efficient* decision procedure *in Type Theory*. Our idea is to use an (untrusted) compiled version of a monadic decision procedure written in Type Theory as an efficient oracle for itself. Like in the certifying style, the decision procedure is developed within an effectful language and used as an oracle by the theorem prover. However, unlike in the certifying style, the decision procedure is written in Type Theory, in a language extended with monads as commonly found in Haskell programs [16]. In this way, programmers have a full set of effects at their hand (references, exceptions, non-termination), together with dependent types to enforce (*partial*) correctness. This decision procedure is then automatically compiled into an impure programming language with an efficient computational model. This compiled code is executed, and a small piece

[1] Usually there is also a previous step where a b is constructed for the given a. This step is called *reification* in the literature.

of information is collected to efficiently *simulate* this execution *in Type Theory* using the initial monadic decision procedure.

To formalize this idea we define the concept of *a posteriori* simulation of effectful computations in Type Theory. Roughly speaking, it involves determining, for a computation C encapsulated in the monadic type $\mathsf{M}\,\mathsf{A}$, the conditions for which there exists a piece of information p such that the evaluation of C, using p, can witness an inhabitant of type A.

We believe this technique to be more lightweight than existing approaches because neither a full proof of correctness nor a certificate checker is required to execute a decision procedure once it is written in our monad.

To sum things up, our contributions are (i) a technique to perform *a posteriori* simulations of effectful computations in Type Theory in order to promote these computations as genuine proofs by reflection; (ii) an informal discussion of different simulable effects; (iii) a plugin[2] for the Coq proof assistant, which enables the effectful computation as an interactive decision procedure of a Coq function written in monadic style; (iv) several examples of proofs by reflection in this new style, showing its simplicity and efficiency.

2 Simulation-Based Proof by Reflection

In this section we give an informal presentation of the simulation-based style of proof by reflection, in Coq. As a running example, we consider the problem of determining if a conjunction of inequalities $\bigwedge_{i\in I} \mathsf{A}_i \leqslant \mathsf{B}_i$ between ground terms of type T logically implies $\mathsf{A} \leqslant \mathsf{B}$ by transitivity, for some A and B. A simple decision procedure for this problem boils down to a depth-first traversal of the graph induced by the hypotheses. In the following procedure implemented in pseudo-code, infinite loops are avoided by marking all of the visited terms:

```
decide (⋀_{i∈I} A_i ⩽ B_i ⇒ A ⩽ B) =
  let rec traverse : T → bool = λC
    if C = B then ⊤
    else if marked C then ⊥
    else
      mark C;
      choice D s.t. ∃j, C ⩽ D ≡ A_j ⩽ B_j ∧ traverse D
    in traverse A
```

This procedure cannot be implemented in Coq as it is, for the simple reason that it uses side effects (marks) and is not obviously terminating (of course, it is, but the argument is not syntactical as Coq requires). As mentioned in the introduction, we are going to implement procedures with side effects using a monad $\mathsf{M}\,\Sigma\,\mathsf{T}'$, where Σ represents the type of the state and T' the returning type of the monad. This procedure is then used as an oracle for itself, as we are going to see in the second part of this section.

Here is our encoding of the decide procedure:

[2] The plugin and the Coq developments of this paper are downloadable online at `http://cybele.gforge.inria.fr`

```
01 Program Definition decide (f: formula) : M Σ (interpret f) :=
02   let (a, b) := goal f in
03   letrec! traverse x [ interpret_hypotheses f → x ⩽ b) ] :=
04     if x =?= b then return (▷ eq_refl x)
05     else if! marked x then error "Not Found"
06     else do! mark x in
07       choice ( f → x ⩽ b) (successors x (hypotheses f))
08         (λ (s : { y : T & interpret_hypotheses f → x ⩽ y }) ⇒
09           let! Hyb := traverse (π1 s) in
10           return (▷ (λ (hs : interpret_hypotheses f) ⇒ le_trans x (π1 s) b))
11         )
12   in ▷ (traverse a).
```

At high level, the code looks like an ML implementation of the pseudo-algorithm, annotated with dependent types. We are going to explain line by line why this procedure is a faithful representation of the pseudo-code shown above, while introducing the notations used in the rest of the paper.

We start by describing the type formula in line 1. It is a record containing a list of pairs of elements (A_i, B_i)—the hypotheses—and a pair of elements (A, B)—the goal. When an element f of this type is interpreted using the function interpret, it produces the type $\bigwedge_{i \in I} A_i \leqslant B_i \to A \leqslant B$. This is the type returned by the monad.

Line 2 is straightforward: it binds the pair of elements being compared in the goal of f to variables a and b. In line 3, the keyword **letrec!** introduces a (potentially nonterminating) recursive function. Behind this syntactic sugar is hidden the application of a dependently-typed general fixpoint operator. The returning type of the local fixpoint traverse is specified between brackets. It returns a proof that the inequality $x \leqslant b$ holds under the hypotheses of formula f. As we can see in line 12, the argument x is instantiated with the element a from the goal, therefore effectively proving $a \leqslant b$.

In line 4 the current element is compared with b, assuming that the type of the elements, T, has decidable equality. If it is equal, then the reflexivity proof is returned using the standard unit monadic combinator **return** [16]. We defer the explanation of the operator ▷.

In line 5, an error is raised if the element x is already marked. We do not show the implementations of functions mark and is_marked (used in next line), but they are straightforward. In line 6 we mark the element, and in lines 7-11 we try to find a proof of $x \leqslant b$ by transitivity, by finding a c such that $x \leqslant c$ and $c \leqslant b$. For that, we make a list with all the *successors* of x, that is, all c such that $x \leqslant c$ is in the list of hypotheses. Then, we call the function choice:

```
01 Fixpoint choice A {T} (cs : list T) (pred : T → M Σ A) : M Σ A :=
02   match cs with
03     | nil ⇒ Error "Not found"
04     | c :: cs ⇒ try! pred c with _ ⇒ choice A cs pred
05     end.
```

The choice operator iterates over a list to find an element c that successfully produces a result using the function pred. At each step of the iteration, the function makes use of the exception mechanism to catch failed attempts and recurse on the tail of the list. Coming back to traverse, in line 9 we call the function recursively using the standard monadic bind operator **let!** $x = e_1$ **in** e_2 [16]. The resulting proof Hyb of c ⩽ b is then used to prove x ⩽ b by transitivity.

Finally, notice that in line 1 we use the standard Coq keyword **Program** [15]. This keyword allows for writing a partial term, where the holes are exposed to the user as proof obligations. In our case, the holes come from type coercions, noted as ⊳, and they are solved automatically by Coq.

The Compiled Decision Procedure as an Oracle. The type system of Coq will not let us apply decide as it is on some formula f to prove the goal. The reason is simple: an infinite loop would lead to breaking soundness of the prover. Instead, in order for decide to be evaluated, it needs some extra information, which we call *prophecy*. For instance, in our example this extra information is the number of steps that leads to a successful result.

To get this information, we execute a compiled version $\mathcal{C}$(decide) in OCaml, which performs the effectful computation. A central property of the system is that $\mathcal{C}(\cdot)$ maps the effectful computations of the monad in Coq to effectful terms in OCaml, in such a way that a relation of *a posteriori* simulation stands between the compiled term $\mathcal{C}(t)$ and the initial monadic term t. Intuitively, if a compiled term $\mathcal{C}(t)$, with t of type M T,[3] converges to a value v, then the same evaluation can be simulated *a posteriori* in Coq, using some prophecy p. This prophecy *completes* computation t in order to get a term convertible to **return** t' for some term t' of type T. We instrument the compiled code $\mathcal{C}(t)$ to produce the prophecy along its execution.

Coming back to our example, the following is the (slightly beautified) extracted OCaml code of the function decide:

```
01 let rec fix f x = incr_nbstep (); f (fix f) x
02
03 let rec choice cs pred0 = match cs with
04   | Nil → failwith "error"
05   | Cons (c, cs0) → try pred0 c with _ → choice cs0 pred0
06
07 let decide f =
08   let (a, b) = goal f in
09   let traverse = fix (fun traverse x →
10       match O.eq_dec x b with
11       | Left → ()
12       | Right → if marked x then failwith "error" else (
13           mark x;
14           choice (successors x (hypothesis f)) (fun s0 → traverse (projT1 s0))
15         ))
16   in traverse a
```

[3] For presentation purposes we leave out the parameter Σ representing the type of the state.

The compiled program has almost the same shape as the source term except that every term in Prop has been erased and that the primitives of the monad are replaced with combinators defined in OCaml. These combinators implement an effect and also contribute in determining the prophecy. For instance, the fix combinator not only implements a general fixpoint but also stores the number of iterations that are performed by the oracle in a global variable.

Once applied to a specific formula, this compiled function may diverge or fail. In the setting of *interactive* theorem proving, divergence is not an important issue because the user stays in front of the screen waiting for an answer, and he or she can always interrupt the oracle if it takes too much time to respond. In the case of a successful execution of the oracle, a prophecy of type nat is extracted from the final value of the mutable cell incremented by fix.

The final proof-term. The resulting proof-term corresponding to the application of the procedure to some formula f is

$$\mathsf{unit\_witness}\,(\mathsf{decide}\,\mathsf{f})\,p\,(\mathsf{refl\_equal}\,\mathsf{true})$$

where p has type Prophecy (in this case, a natural number), and for any type T,

$$\begin{aligned}\mathsf{unit\_witness} &: \forall \mathsf{x} : \mathsf{M}\,\mathsf{T}, \mathsf{Prophecy} \to \mathsf{is\_unit}\,\mathsf{x} = \mathsf{true} \to \mathsf{T}\\ \mathsf{is\_unit} &: \mathsf{M}\,\mathsf{T} \to \mathsf{bool}\end{aligned}$$

The execution time of checking that this term has type interpret f is split between the execution time of typechecking the prophecy p and the weak head normalization of the procedure, using p to guide the reduction. The overall execution time of the proof-by-reflection results from executing the decision procedure in OCaml plus typechecking the final proof-term, which as we just mentioned, essentially consists of executing the decision procedure a second time in Coq.[4] One can wonder if it is not a waste of time to execute the decision procedure twice, but, as it turns out, using the hints in p, the execution time of the simulation can be tremendously reduced in comparison with the execution of the oracle. This optimization is the subject of Section 5.2.

Putting all the pieces together, our plugin performs the following steps when proving a goal with a monadic procedure proc: (1) Translates and compiles proc into OCaml. (2) Executes the compiled code $\mathcal{C}(\mathsf{proc})$ and obtains prophecy p. (3) Builds proof term unit_witness proc p (refl_equal true). Notice that the proof developer only has to develop the procedure.

3 *A Posteriori* Simulation of Effects

In this section we formalize the principle of *a posteriori* simulation of effectful computations. The interested reader is invited to read the proofs from the companion technical report [6]. In order to promote a clear formalization we

[4] We assume compilation time not to be significant.

will focus only on simply typed λ-calculus, but the results presented are easily extensible to full Type Theory and OCaml, for the pure and impure calculus, respectively. More precisely, we define two languages: λ, a purely functional and strongly normalizing programming language with monadic constructs, and $\lambda_{v,\perp}$, a non-terminating functional programming language. The definition of λ is parameterized by a monad M, which is abstractly specified by a set of requirements. Accordingly, $\lambda_{v,\perp}$ offers impure operators that match the effectful primitives of the monad M.

Conventions. We write $\overrightarrow{e}$ for a sequence $e_1\, e_2\, \ldots\, e_n$ where $n \geqslant 0$. If a function $\mathcal{F}$ is defined over e then we abusively write $\mathcal{F}\,(\overrightarrow{e})$ for the pointwise extension of $\mathcal{F}$ to a sequence of e. For the sake of conciseness, we often omit universal quantifiers in types when they appear in outermost prenex position.

3.1 λ, a Purely Functional Language

The language λ is the simply typed λ-calculus *à la Curry* with constants:

$$\begin{array}{ll} t,\, u ::= x \mid \lambda x.t \mid t\,t \mid \mathbf{c} & \mathsf{T} ::= \mathsf{T} \to \mathsf{T} \mid \mathsf{C}\,\overrightarrow{\mathsf{T}} \\ \mathbf{c} ::= \mathsf{unit} \mid \mathsf{bind} \mid \Downarrow \mid \nabla & \mathsf{C} ::= \mathsf{M} \mid \mathsf{P} \end{array}$$

Constants include the usual monadic combinators for effects in the spirit of [16]: unit lifts a term of type T as a computation of type $\mathsf{M}\,\mathsf{T}$, and bind composes two computations. Effectful primitives of the monad are kept abstract by regrouping them in the syntactic category ∇. The types include functional types and type constructor applications which are assumed well-formed. M and P are the type constructors for monad and prophecy, respectively. We omit the typing rules but they are standard.

The constant $\Downarrow$ and the type constructor P are unusual. The role of $\Downarrow$ is to perform *a posteriori* simulation using a value p of type P produced by the oracle. We read $\Downarrow_p\, t$ as "the reduced computation of t using the prophecy p". We require the existence of a total order $\leqslant$ over values of type P and a minimal element $\perp$ for this order. A reduced computation is still a computation, so $\Downarrow$ has type $\mathsf{P} \to \mathsf{M}\,\mathsf{T} \to \mathsf{M}\,\mathsf{T}$.

We are interested in reasoning on $\beta\delta$-convertibility between terms (where the δ-reduction is the unfolding of constant definitions). We write $\star t$ for $\Downarrow_\perp \mathsf{unit}\; t$ and we say that a computation has converged if there exist a prophecy p and a term t' such that $\Downarrow_p t$ is convertible to $\star t'$.

Finally, the standard notion of monad is extended with a mechanism of simulation directed by a prophecy.

Definition 1 (Simulable monad). *A type constructor M is a simulable monad if it is equipped with unit, bind, $\Downarrow$ and an associated type for prophecies P, such that the requirements 1, 2, 3 and 4 are fulfilled.*

Requirement 1 (Standard monadic laws)

$$\begin{aligned} \mathit{bind}\,(\mathit{unit}\ t)\ f &= f\ t \\ \mathit{bind}\,t\ (\lambda x.\mathit{unit}\ x) &= t \\ \mathit{bind}\,(\mathit{bind}\,t_1\ t_2)\ t_3 &= \mathit{bind}\,t_1\ (\lambda x.\mathit{bind}\,(t_2\ x)\ t_3) \end{aligned}$$

Requirement 2 (Reduction)

$$\begin{aligned} &\forall\ t, p_1, p_2,\ \Downarrow_{p_1} \mathit{unit}\ t = \Downarrow_{p_2} \mathit{unit}\ t. \\ &\forall\ t, u, p_1, p_2,\ p_1 \leqslant p_2\ \mathit{and}\ \Downarrow_{p_1} t = \star u\ \mathit{implies\ that}\ \Downarrow_{p_2} t = \star u. \\ &\forall\ p,\ \Downarrow_p \mathit{bind}\,t_1\ t_2 = \Downarrow_p \mathit{bind}\,(\Downarrow_p t_1)\ t_2 \end{aligned}$$

3.2 $\lambda_{v,\bot}$, A Call-By-Value Impure Functional Language

The impure functional and non-terminating language $\lambda_{v,\bot}$ has the same syntax as λ, except that now constants only consist of effectful operators. The language is equipped with an *instrumented* big-step operational semantics for a weak call-by-value reduction strategy. The executions are carried out under environments η assigning closed values v to variables: $\eta ::= \cdot \mid \eta; x \mapsto v,\ v ::= \boldsymbol{c}\overrightarrow{v} \mid (\lambda x.u)\,[\eta]$. Closed values comprise full applications of effectful constants to values and closures. The judgment in this instrumented semantics is $\eta \vdash u \Downarrow_{p \to p'} v$, which is intended to be read as "the execution of a term u under the environment η converges to a value v and computes a prophecy p' from an initial prophecy p". We keep abstract the rules for constants: they will be characterized by the requirement 4.

$$\text{R-Var}\ \frac{}{\eta \vdash x \Downarrow_{p \to p} \eta(x)} \qquad \text{R-Lam}\ \frac{}{\eta \vdash \lambda x.u \Downarrow_{p \to p} (\lambda x.u)\,[\eta]}$$

$$\text{R-App}\ \frac{\eta \vdash u_1 \Downarrow_{p \to p_1} (\lambda x.u)\,[\eta'] \qquad \eta \vdash u_2 \Downarrow_{p_1 \to p_2} v_1 \qquad \eta'; x \mapsto v_1 \vdash u \Downarrow_{p_2 \to p'} v}{\eta \vdash u_1\ u_2 \Downarrow_{p \to p'} v}$$

The purpose of the instrumentation of the compiled code is to monotonically refine the prophecy at each step of the computation:

Requirement 3 (Monotonicity of prophecy computation)

$$\forall\ p,\ p',\ \eta \vdash u \Downarrow_{p \to p'} v\ \mathit{implies}\ p \leqslant p'.$$

Compilation Now, we define the compilation function $\mathcal{C}(\cdot)$ from λ to $\lambda_{v,\bot}$.

$$\begin{array}{lll|l} \mathcal{C}(x) = x & \mathcal{C}(\mathsf{unit}) = \lambda x.x & & \mathcal{C}(\mathsf{M}\,\mathsf{T}) = \mathcal{C}(\mathsf{T}) \\ \mathcal{C}(\lambda x.t) = \lambda x.\mathcal{C}(t) & \mathcal{C}(\mathsf{bind}) = \lambda x, y.\,y\,x & & \mathcal{C}(\mathsf{C}\ \overrightarrow{\mathsf{T}}) = \mathsf{C}\,(\mathcal{C}(\overrightarrow{\mathsf{T}})) \\ \mathcal{C}(t_1\,t_2) = \mathcal{C}(t_1)\mathcal{C}(t_2) & \mathcal{C}(\Downarrow_p) = \text{undefined} & & \mathcal{C}(\mathsf{T}_1 \to \mathsf{T}_2) = \mathcal{C}(\mathsf{T}_1) \to \mathcal{C}(\mathsf{T}_2) \end{array}$$

The translation replaces the monadic constructs unit and bind with their respective definitions in the identity monad, and converts each effectful primitive

of the monad to the corresponding impure construction of $\lambda_{v,\bot}$. The type for prophecies is kept fully abstract to the programmer. As a consequence, only the instrumented compiled code is allowed to generate prophecies. Therefore, the compilation of $\Downarrow_p$ is explicitly *undefined* because this operator cannot appear in a well-typed user-written monadic term.

The compilation of an effectful monadic constant must extend the prophecy in a sufficient way to make the simulation converge.

Requirement 4 (Adequate instrumented compilation) $\forall p_0, \ldots, p_{n+1}, p,$

$$\textit{if} \begin{cases} \forall i,\ \eta \vdash \mathcal{C}(t_i) \Downarrow_{p_i \to p_{i+1}} v_i \\ \qquad \eta \vdash \mathcal{C}(\boldsymbol{c}\,(t_0, \ldots, t_n)) \Downarrow_{p_0 \to p} v \end{cases} \qquad \textit{then}\ \ \exists u, \Downarrow_p \boldsymbol{c}\,(t_0, \ldots, t_n) = \star u$$

3.3 Examples of Simulable Monads

The "Trace" Prophecy. Given a monad M with an underlying effectful computation model specified by a reduction relation, there is always a prophecy to simulate a converging effectful reduction: the reduction chain itself. However, such a naive implementation of prophecies is obviously inefficient.

Non-Termination and Partiality. The type $\mathsf{nat} \to \mathsf{option}\,\mathsf{T}$ defines an adequate monad to represent non-terminating computations of type T. A general fixpoint operator is defined by induction over the input natural number. If the number of iterations is sufficient then the computation produces a term $\mathsf{Some}\,\mathsf{t}$, otherwise None. For this monad, the natural type for prophecies is nat and the instrumentation only has to compute an over-approximation of the number of iterations for all the fixpoints of the program. Therefore, a single global variable is enough to represent the prophecy.

State. The type $\mathsf{state} \to \mathsf{T} \times \mathsf{state}$ defines an adequate monad to model stateful computations. A state monad is naturally simulable without a need for prophecies because the operations read and write are total. Yet, if the monad also provides an operation ref to dynamically allocate mutable references, it is hard to ensure statically that a given reference belongs to the state. In that case, the state monad has to be composed with the partiality monad and inherit its type for prophecies. Furthermore, the prophecy can also embed the initial state used to evaluate the monadic term: this is an opportunity to import some precomputed results from the oracle (see Section 4).

Non-determinism. The type $\mathsf{list}\,\mathsf{T}$ defines an adequate monad to model nondeterministic computations, which is useful in proof search procedures. An important operator of this monad is choice of type $\mathsf{M}\,\mathsf{T} \to (\mathsf{T} \to \mathsf{M}\,\mathsf{T}') \to \mathsf{M}\,\mathsf{T}'$, a partial function that picks an arbitrary choice in all the possibilities. If the list is empty, there is no such choice. But, if a computation had converged, there exist a list of choices that leads to a result. An interesting prophecy is exactly this list of choices (see Section 5.2).

3.4 A Posteriori Simulation

The main theorem states that, if the evaluation of $\mathcal{C}(t)$ converges for some computation t, then there exists a prophecy p to simulate t back in λ.

Theorem 1 (A posteriori simulation). *Let* $\cdot \vdash t : M\,T$ *a computation which compilation converges to a value, that is* $\cdot \vdash \mathcal{C}(t) \Downarrow_{p \to p'} v$ *holds. Then there exists a term* t' *such that* $\Downarrow_{p'} t = \star t'$.

4 Implementation

We provide a plugin for Coq to develop proofs using the method described in this work. The plugin includes (i) a library with the definition of a simulable monad to write effectful decision procedures; (ii) a tactic called coq waiting for a monadic term t of type M T to try to solve a goal T. Behind the scene, the tactic compiles the monadic term into an OCaml program, executes this program and if its execution converged, uses the resulting prophecy to produce a proof-term in Coq.

The formal notion of simulable monad served as a guideline for the implementation: we defined a compilation function from Coq to OCaml as well as a simulable monad in Coq that respect the requirements drawn by our formal study. However, to improve the usability and the efficiency of our tool, some practical aspects of the implementation differ from the formal specification.

4.1 A Simulable Monad in Coq

Our monad combines[5] a partiality monad, a non-termination monad, a state monad and a printing monad. It is still possible to implement nondeterminism in it, as we will see in the example from Section 5.2. The monad is parametrized by a signature Σ to type the memory (see below). Its type definition is:

$$\mathsf{M}\ \Sigma\ \alpha = \mathsf{State.t}\ \Sigma \to (\alpha + \mathsf{string}) \times \mathsf{State.t}\ \Sigma$$

The monad takes a state and returns a new state plus a value of type α if the computation is successful, or an error message in case of failure. The state is implemented as a dependent record containing: (i) the number of steps allowed in recursion, (ii) a list of messages (used for debugging by the printing monad), and (iii) the memory.

The size of the memory has to be dynamic, but at the same time the memory has to be statically typed. Our solution is to parametrize the memory by a signature Σ, containing the exact list of types $T_1, T_2, \ldots, T_n$ that will be used. Then, the memory is a list of n regions of types $T_1, \ldots, T_n$ respectively. The content of a region is unbounded. For instance each region may contain a list of elements. A reference has type Ref.t Σ T_i, and its implementation is simply the

[5] In this work, unlike in Haskell, we are not interested in a fine grain control of effects so we provide only one monad with all the effectful operations we found useful.

natural number i corresponding to the i-th type in the signature Σ. All in all, here are the effectful operations offered by the monad:

$$\begin{array}{ll}
\mathsf{ref} & : T_i \to \mathsf{M}\ \Sigma\ (\mathrm{Ref.t}\ \Sigma\ T_i) \\
\mathsf{read} & : \mathrm{Ref.t}\ \Sigma\ T \to \mathsf{M}\ \Sigma\ T \\
\mathsf{write} & : \mathrm{Ref.t}\ \Sigma\ T \to T \to \mathsf{M}\ \Sigma\ () \\
\mathsf{print} & : \alpha \to \mathsf{M}\ \Sigma\ ()
\end{array}
\qquad
\begin{array}{ll}
\mathsf{error} & : \mathsf{string} \to \mathsf{M}\ \Sigma\ \alpha \\
\mathsf{try_with} & : (() \to \mathsf{M}\ \Sigma\ \alpha) \to \\
 & \quad (\mathsf{string} \to \mathsf{M}\ \Sigma\ \alpha) \to \mathsf{M}\ \Sigma\ \alpha \\
\mathsf{dependentfix} & : (\mathcal{F} \to \mathcal{F}) \to \mathcal{F} \\
 & \quad \text{with } \mathcal{F} = \forall (x : A).\mathsf{M}\ \Sigma\ (B\ x)
\end{array}$$

Pre-computation. The memory is partitioned into two parts: InputMem and TmpMem. TmpMem is initially empty and corresponds to the memory in the usual state monad. InputMem is initialized by the OCaml program and given as the initial (read-only) memory to Coq as a prophecy. Roughly speaking, the order on the prophecies is induced by the distance between the contents of this initial memory and the information needed to compute the same result in Coq as in OCaml *i.e.* the required number of fixpoint iterations and the values that were pre-computed in OCaml. Inside the implementation of the monad, this forces us to program differently for these two environments and, for this reason, we defined a low-level internal operator, select, of type $\forall\, \alpha, (() \to \alpha) \to (() \to \alpha) \to \alpha$ which is defined in Coq as $\mathsf{select}(f, g) = f()$ and compiled in OCaml as $\mathcal{C}(g())$. For more information about select, we defer the reader to Section 5.2. To fulfill the requirements to ensure that our monad is simulable, we make sure that the OCaml version of each operator only refines the contents of the InputMem during its effectful execution.

4.2 In OCaml

The compilation of a monadic term written in Coq to a program in OCaml is implemented by customizing the existing extraction mechanism of Coq [13], where the new monadic constructs are extracted as follows:

$$\begin{array}{rl}
\mathsf{M}\ \Sigma\ \alpha & \mapsto \alpha \\
\mathsf{unit} & \mapsto \mathbf{fun}\ x \to x \\
\mathsf{bind} & \mapsto \mathbf{fun}\ x\ f \to f\ x \\
\mathsf{print} & \mapsto \mathbf{fun}\ x \to \mathrm{print_endline}\ x \\
\mathsf{dependentfix} & \mapsto \mathbf{let\ rec}\ \mathsf{fix}\ f = \mathbf{fun}\ x \to \\
 & \qquad \mathrm{incr_nbsteps}(); f\ (\mathsf{fix}\ f)\ x \\
 & \quad \mathbf{in}\ \mathsf{fix}\ f\ x
\end{array}
\qquad
\begin{array}{rl}
\mathsf{error} & \mapsto \mathbf{fun}\ x \to \mathrm{failwith}\ x \\
\mathsf{try_with} & \mapsto \mathbf{fun}\ f\ h \to \mathbf{try}\ f\ () \\
 & \quad \mathbf{with}\ s \mapsto h\ s \\
\mathsf{tmp_ref} & \mapsto \mathbf{fun}\ i\ v \to \mathrm{ref}\ v \\
\mathsf{input_ref} & \mapsto \mathbf{fun}\ i\ v \to \mathrm{register_ref}\ i\ v \\
\mathsf{read} & \mapsto \mathbf{fun}\ r \to !r \\
\mathsf{write} & \mapsto \mathbf{fun}\ r\ v \to r := v
\end{array}$$

Since we are using the built-in effectful mechanisms provided by OCaml, the monad is converted into the identity monad and, thus, the bind and unit combinators are defined accordingly.

The print and partiality monad are implemented with the standard print function and exceptions. The fixpoint operator adds instrumentation to count the number of iterations in a global variable. The memory operators are handled by OCaml's references. References are divided into tmp_ref and input_ref. The first ones are just normal OCaml's references, while the second ones are registered

in an array, using the function register_ref. Thus, we can collect the values of all the input references at the end of the execution to pass them to Coq.

4.3 Communication from OCaml to Coq

Once the execution of the OCaml code is done, we generate the prophecy for Coq. It contains two parts: the number of steps and the memory in InputMem. The first part is easy to communicate back, as it is just a natural number. As for InputMem, it is more tricky since we need to reify OCaml data into Coq terms. Notice that this is not possible in general, for example for abstractions or for proof terms, since the extraction to OCaml erases too much information from the source term. Our solution is to provide an ad-hoc reification mechanism using binary trees: for every type T in the input memory signature, the user needs to provide a morphism between T and a binary tree.

5 Examples

We now show examples of Coq programs written using a simulable monad. The first example describes how to write effectful programs, while the second example illustrates how the performance of an algorithm is greatly improved by using compilation to OCaml and cross-stage memoization.

5.1 Congruence-Closure

The congruence-closure problem is about proving equality of two first-order terms, given a set of known equalities. It can be solved efficiently using the union-find algorithm [2]. In [7], a reflexive version of the algorithm is presented, which is purely functional and proven correct. A large part of the code is devoted to prove termination and implementing functional arrays. We wrote this algorithm in our system using the partiality monad to avoid proving termination. We focus on the `Find` function:

```
01 Program Definition Find hash u : M Σ {u' : Index.t | u ≡ u'} :=
02   dependentfix (λ i ⇒ {j : Index.t | i ≡ j}) (λ find i ⇒
03     let! eq_proof := MHash.Read hash i in
04     let (i', j, Hij) := eq_proof in
05     if i ≡ i' then (* case i = i': should always be the case *)
06       if i ≡ j then (* case i = j: we find it *) return (exist _ j Hij)
07       else (* case i <> j: we have to continue from j *)
08         let! r := find j in
09         let (k, Hjk) := r in
10         do! MHash.Write hash i (EqProof.Make (i := i) (j := k) _) in
11         return (exist _ k _)
12     else (* case i <> i': unexpected *) error "Find: i ≠ i'")
13   u.
```

At high level this function retrieves the representative u' of the equivalent class of u, along with a proof of the equality among u and u'. It iterates over a hash-table `hash` from expressions (`Index.t` in the code) to expressions, crawling the hash table until an element points to itself. If that is the case, then we reach the representative. The hash table also contains the proof of equality, which is used transitively to compute the resulting equality proof.

Programming with Effects in Coq. Proving termination of the algorithm is hard since it requires to maintain the invariant that the table is not cyclic. Luckily, we are exempt to do such proof, thanks to the `dependentfix` operator that allows for non-termination. The hash-table is a mutable structure with a `read` and a `write` operation. It is implemented as a mutable map from expressions to expressions, with an additional proof of equality.

Dependently-Typed Programming with Partial Functions. We keep the power of the Coq type system despite the fact that we are working in a monad. The `Find` function has a dependent type specifying that the result is the representative term u', equal to the input term u. The proof term is generated in the monad, so we can rely on run-time checks, which may fail, instead of proving invariants. For example the invariant $i = i'$ holds but does not have to be statically proven. Instead it is checked dynamically (comparison of i and i' on line 5). The result is used to coerce a proof of $i' = j$ to $i = j$ (done automatically by the `Program` command in our example). If the check fails, we raise an exception handled by the partiality monad. In this way we can partially specify our programs. Notice that we are not forced to use partial programs, we can also use pure Coq functions leading to stronger static guarantees. This flexibility is not available in mainstream functional languages like OCaml.

5.2 A Tactic for Lattices

James and Hinze [12] present a reflection-based tactic to solve lattice (in)equalities based on the algorithm proposed by Whitman [17], which is known for being exponential in the worst case. In this work, the authors made the following remark:

> "Possible future work is to turn our current implementation [...] into one that uses dynamic programming to memoize the recursive calls. However, this is not a trivial task. Coq's programming language is purely functional [...], so any data-structure that we use for memoization must be purely functional and operations on that data-structure must all be proved terminating."

In this section we provide a tactic similar to James and Hinze's, but that uses memoization. In our case, unlike in the recommendation made in the quoted text, we use a form of cross-stage memoization to remember the successful path of execution made by the OCaml version and to transmit it to the Coq version. In this way, the exponential algorithm is executed only in OCaml, while Coq just recreates the successful path made by the OCaml version mimicking what a certificate checker would do. Unsurprisingly, the implementation presented here greatly outperforms the one presented by James and Hinze. For details, we refer to the original work cited above or to the accompanying code.

Whitman's algorithm. The algorithm, as written by James and Hinze with some simplifications and syntax sugaring follows.

```
01 Program Fixpoint leq (t u : Term) : {b : bool | b → t ⩽ u} :=
02   match (t,u) with
03   | (Var m, Var n) ⇒ m ≡ n
04   | (Join t1 t2, u) ⇒ leq t1 u ∧ leq t2 u
05   | (t, Meet u1 u2) ⇒ leq t u1 ∧ leq t u2
06   | (Var m, Join u1 u2) ⇒ leq t u1 ∨ leq t u2
07   | (Meet t1 t2, Var n) ⇒ leq t1 u ∨ leq t2 u
08   | (Meet t1 t2, Join u1 u2) ⇒ leq t1 u ∨ leq t2 u ∨ leq t u1 ∨ leq t u2
09   end.
```

What is important to notice is the $\vee$ branching in the last three cases. In particular, the last case requires the algorithm to branch four times! This is the culprit for the exponential time taken by the algorithm in some examples.

Remembering the past. To simulate only the interesting part of the proof search, the simulation must choose the right side of every disjunction. This optimization lies on the following function which advantageously replaces $\vee$:

```
01 Fixpoint tryBranches (ref: Ref.t Σ _) (n_branch: nat) (k : TermPairMap.key) B
02   (branches: list (unit → M Σ B)) : M Σ B := select
03   (* Coq *) (λ _ ⇒ let! map := !ref in
04       let! n_branch := extract_some (TermPairMap.find k map) in
05       let! branch := extract_some (nth_error branches n_branch) in branch tt)
06   (* OCaml *) (λ _ ⇒ let! map := !ref in
07       match TermPairMap.find k map with
08       | Some n ⇒ let! branch := extract_some (nth_error branches n) in branch tt
09       | None ⇒ match branches with
10         | nil ⇒ error "No branch left to try"
11         | branch :: branches' ⇒ try!
12             let! r := branch tt in
13             let! map := !ref in
14             do! ref :=! TermPairMap.add k n_branch map in
15             return r
16           with _ ⇒ tryBranches ref (S n_branch) k branches'
17         end
18       end).
```

This function uses the select operator to behave differently in OCaml than in Coq. In OCaml, it tries to execute the code in all of the branches, and the returned value comes from the first branch succeeding in its execution. In addition, the position of the successful branch is added to the map referenced by ref. If no branch succeed, then it raises an error. Before exploring the branches, it first checks whether it is known which branch to take, and if this is the case, it executes the code from that branch only. In Coq, it first reads the position from the map, and executes only the code from the branch in this position. In both cases, the key k used to store the position in the map is given as a parameter.

This key is instantiated with a pair containing the terms from both sides of the inequality under consideration. The Whitman's algorithm is changed to take advantage of the branching function just described.

There is a couple of important remarks that must be made about this optimization. First, the very powerful select operator can obviously break the theoretical requirements to achieve the *a posteriori* simulation. It is provided to allow the user to create primitive operators not present in the monad. Second, our implementation of the non determinism assumes that there is no side-effect in the failing branches that may affect the successful ones.

Performance. As expected, we get a great performance gain, shown in Figures 1 and 2. These plots show the time it takes Coq to typecheck the result, for two different classes of problems. The time to typecheck the result from the original purely functional algorithm is shown in rounded dots, while for the effectful code it is shown in squares. In Figure 1 we consider a problem with an increasing number of variables, where there is no repetition in the formula (therefore every combination should be taken into account). In Figure 2, we increment the number of times a certain pattern occurs in an inequality, showing how our method benefits from reusing previously computed paths. To sum things up, these plots clearly shows the benefit of using prophecies to help the typechecker save some computation.

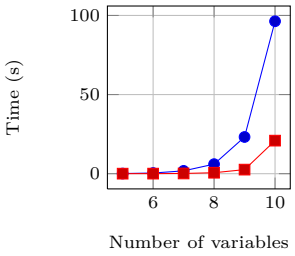


Fig. 1. Typechecking time for exponential proof terms

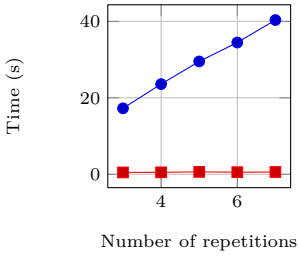


Fig. 2. Typechecking time for terms with a repetitive pattern

6 Related Work

Extending Coq with imperative features. Coq has been extended with imperative features [1]. The methodology behind this extension is to offer to Coq's user a functional interface to data structures that are efficiently compiled internally. This solution is transparent to the user: there is no need to write decision procedures in a monad to use imperative mechanisms. Yet, the trusted base, *i.e.* the kernel of Coq, had to be extended. Actually, the two systems can be used together: we could make use of the efficient data structures provided by this extension to define some of the effectful operators of our monad improving the performance of the *a posteriori* simulation done at Qed time.

Prophecies in Type Theory. Several works propose [5,14] methods to define and to reason on general recursive functions in Type Theory. Bove and Capretta [5] formally define a notion of prophecy, a coinductive predicate derived from a set of non-overlaping recursive equations characterizing the co-domain of the partial function defined by these equations. Our prophecies and Bove and Capretta's share the same role of prediction. However, our prophecies do not need to be co-inductive because our monad uses them in direct style. Besides, our prophecies are computed outside Coq by an efficient computational model *i.e.* OCaml.

7 Conclusion

In this paper, we presented a novel technique to write decision procedures in Coq. We described its implementation as a plugin and we hope that it will simplify the development of proofs by reflection in the future.

Acknowledgments. We would like to thank the anonymous reviewers for their thorough and helpful reviews.

References

1. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with Imperative Features and its Application to SAT Verification. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, Heidelberg (2010)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press (1998)
3. Blech, J.O., Grégoire, B.: Certifying compilers using higher-order theorem provers as certificate checkers. FMSD (2011)
4. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Ito, T., Abadi, M. (eds.) TACS 1997. LNCS, vol. 1281, pp. 515–529. Springer, Heidelberg (1997)
5. Bove, A., Capretta, V.: Computation by prophecy. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 70–83. Springer, Heidelberg (2007)
6. Claret, G., González Huesca, L., Régis-Gianas, Y., Ziliani, B.: Lightweight proof by reflection using a posteriori simulation of effectful computation. Technical report (2013), `http://cybele.gforge.inria.fr/download/cybele_technical_report.pdf`
7. Corbineau, P.: Autour de la clôture de congruence avec coq. Master's thesis, ENS (2001) (in French)
8. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In: ICFP (2011)
9. Gordon, M.J., Wadsworth, C.P., Milner, R.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
10. Grégoire, B., Pottier, L., Théry, L.: Proof certificates for algebra and their application to automatic geometry theorem proving. In: Sturm, T., Zengler, C. (eds.) ADG 2008. LNCS, vol. 6301, pp. 42–59. Springer, Heidelberg (2011)
11. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical report, SRI Cambridge (1995)

12. James, D.W.H., Hinze, R.: A Reflection-based Proof Tactic for Lattices in Coq. In: TFP (2009)
13. Letouzey, P.: Extraction in Coq: An Overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008)
14. Pichardie, D., Rusu, V.: Defining and Reasoning About General Recursive Functions in Type Theory: A Practical Method. Research report, IRISA (2005)
15. Sozeau, M.: Subset coercions in coq. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007)
16. Wadler, P.: Comprehending monads. MSCS (1992)
17. Whitman, P.: Free Lattices. Harvard University (1941)

Automatic Data Refinement

Peter Lammich

Technische Universität München
lammich@in.tum.de

Abstract. We present the Autoref tool for Isabelle/HOL, which automatically refines algorithms specified over abstract concepts like maps and sets to algorithms over concrete implementations like red-black-trees, and produces a refinement theorem. It is based on ideas borrowed from relational parametricity due to Reynolds and Wadler.

The tool allows for rapid prototyping of verified, executable algorithms. Moreover, it can be configured to fine-tune the result to the user's needs. Our tool is able to automatically instantiate generic algorithms, which greatly simplifies the implementation of executable data structures.

Thanks to its integration with the Isabelle Refinement Framework and the Isabelle Collection Framework, Autoref can be used as a backend to a stepwise refinement based development approach, having access to a rich library of verified data structures. We have evaluated the tool by synthesizing efficiently executable refinements for some complex algorithms, as well as by implementing a library of generic algorithms for maps and sets.

1 Introduction

If one wants to generate efficiently executable code for an algorithm verified in Isabelle/HOL, there are currently two alternatives. The first alternative is to do the formalization with executability in mind, e. g. using lists instead of sets. Then, Isabelle/HOL's code generator [7,9] can extract executable code from the formalization in various functional target languages like ML and Scala. However, being limited to only executable concepts in the formalization has the disadvantage of cluttering the proofs with implementation details. This makes the proofs more complicated, and may even render proofs of medium complex algorithms unmanageable.[1] Moreover, changing the implementation later means essentially redoing the whole formalization.

A well known solution to this problem is refinement [10], in particular refinement calculus [1,2]. Here, an algorithm is formulated and proven correct on an abstract level, and then refined towards an efficient implementation in possibly multiple refinement steps. As each refinement step preserves correctness, the resulting algorithm is correct. Stepwise refinement simplifies the proofs by modularization: The correctness proof of the abstract algorithm focuses on the algorithmic idea, not caring about implementation, while the proof of a refinement step shows the correctness of the implementation of particular abstract concepts, not caring about the overall correctness.

[1] The author had this experience with early versions of algorithm formalizations [17,24].

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 84–99, 2013.

In the context of code extraction from Isabelle/HOL, several approaches to data refinement have been explored [15,20]. They focus on the special case of *pure data refinement*, where abstract types (e. g. sets) are replaced by concrete implementations (e. g. red-black trees), but the structure of the algorithm is preserved. Conceptually, this refinement is simple: Rephrase the algorithm using efficient implementations for the abstract concepts and prove that it refines the original algorithm. However, using existing techniques, this still requires much effort. In particular, to produce multiple implementations from the same abstract algorithm, it has to be manually rephrased for each implementation. An alternative is to set up a parameterized version of the algorithm, e. g. by using locales. However, this approach suffers from limited polymorphism in typical HOL theorem provers (cf. [12]).

In this paper, we present the Autoref tool, which performs pure data refinement automatically. Given an algorithm phrased over abstract concepts like sets and maps, it automatically synthesizes a concrete, executable algorithm and the corresponding refinement theorem. It has heuristics that try to choose suitable implementations by default. Moreover, the defaults can easily be overridden by the user. Thus, it can be used for both rapid prototyping and generating the final, fine-tuned version of the code.

Autoref is based on the idea of relational parametricity [25,26], which is used to express data refinement for higher-order types.

To make it applicable for the development of actual algorithms, Autoref is integrated with the Isabelle Refinement Framework [19,18] and the Isabelle Collection Framework [16,15]. The former supports a development approach based on stepwise refinement, and the latter provides a large collection of verified data structures. Both tools have already been used for successful verification of complex algorithms [17,19,5,6]. Using Autoref as a back end greatly simplifies this development process. As a case study, we have applied Autoref to generate executable code for a nested depth-first search algorithm and an algorithm to compute simulation relations on finite state machines.

Another distinguishing feature of Autoref is its support for generic programming [21] in a user-transparent manner. During the synthesis, the concrete implementation of an abstract operation may be synthesized via a generic algorithm. To demonstrate this feature, we have developed a library of generic map and set algorithms.

Moreover, we provide implementations of data structures that overcome some limitations of the implementations provided by the Isabelle Collection Framework. Using Autoref's support for parametricity reasoning, we were able to generalize the existing implementations without redoing their correctness proofs.

The implementation of Autoref and the case studies are available at `https://www21.in.tum.de/~lammich/autoref`.

Related Work We already mentioned the concepts of data refinement [10], refinement calculus [1], and parametricity [25,26] that underly Autoref, as well as various manual approaches to data refinement [15,20].

The transfer package [13] for Isabelle/HOL can automatically transfer theorems over quotient types. It is also based on parametricity, and inspired us to use this technique in Autoref.

Parallel to our work, support for automatic data refinement has been integrated into the Isabelle/HOL code generator by Haftmann et al. [8], and also the verified code generator for HOL4 of Myreen et al. [22] supports automatic data refinement. This work is complementary to ours, and we provide a detailed comparison in Section 4.5.

The remainder of this paper is organized as follows: Section 2 describes the basic ideas that underly our tool. Section 3 describes how to implement a usable tool based on these ideas. Section 4 reports on our case studies, and, finally, Section 5 gives a short conclusion and outlook to future work.

2 Basic Ideas

In this section, we describe the basic ideas behind the Autoref tool. After a short introduction to Isabelle/HOL (§2.1), we describe relators (§2.2) and transfer rules (§2.3), and, finally, our treatment of equality and type classes (§2.4).

2.1 Isabelle/HOL

Autoref is implemented in Isabelle/HOL [23], an LCF-style theorem prover for higher order logic. However, the same approach could also be implemented within other HOL theorem provers. We assume the reader has basic knowledge of HOL-style theorem provers. In this subsection, we only describe some aspects of Isabelle/HOL that are essential for this paper.

A type in Isabelle/HOL is either a type variable or a type constructor applied to a list of types. Type variables are written with leading ticks, e. g. *'a*, and application of a type constructor is written in postfix notation, e. g. *'a list* or *('a,'b)prod*. Moreover, there is syntactic sugar for some standard types: the function type $'a \to 'b$, the product type $'a \times 'b$, and the sum type $'a + 'b$.

A term in Isabelle/HOL is either a constant, a variable, a bound variable, function application, or λ-abstraction. Constants and variables are annotated with their type, and λ-abstractions are annotated with the parameter type.

In Isabelle/HOL, there is a further distinction between schematic and free type-/term variables. Schematic variables can be instantiated by unification, while fixed variables cannot. Schematic variables are denoted by a leading question mark, e. g. *?'a* or *?a*.

Schematic variables can be used for synthesis: For example, when starting with a proof goal of the form $?a = 1$, $?a$ may be instantiated during the proof. If we resolve the above goal with reflexivity, $?a$ is instantiated to 1, and the theorem that is proved is $1 = 1$. In contrast, free variables cannot be instantiated during the proof. However, they are converted to schematic variables after the proof has been finished. Thus, reflexivity is stated as the goal $x{=}x$, and later gets converted to $?x{=}?x$. However, by convention, we do not use question marks for variables when referring to a theorem.

Isabelle/HOL has no naming conventions to distinguish free variables from constants. In this paper, the distinction between variables and constants should always be clear from the context.

2.2 Relators

In order to refine an abstract program to an executable, concrete one, all types and operations in the abstract program must be refined to concrete counterparts that are executable. For example, a set in the abstract program may become a red-black tree in the concrete program, and insertion into a set may become insertion into a red-black tree.

We use relators [3] to express the relationship between concrete and abstract types. Let T_A be an $n$ary type constructor, and let T_C be its concrete version. Then, a *relator*[2] R_T between T_C and T_A is an $n$ary function that maps relations between concrete and abstract argument types to a relation between T_C and T_A:

$$R_T :: ('c_1 \times 'a_1)set \to \ldots \to ('c_n \times 'a_n)set \to (('c_1,\ldots,'c_n)T_C \times ('a_1,\ldots,'a_n)T_A)set$$

We use the postfix notation $\langle R_1,\ldots,R_n\rangle R_T$ for relators, to make them similar to the notations $('c_1,\ldots,'c_n)T_C$ and $('a_1,\ldots,'a_n)T_A$ for the corresponding types.

A *natural* relator relates a type constructor to itself, not changing the shape of the values.

Example 1. Consider the list type *'a list ::= Nil | Cons 'a ('a list)*. The natural relator for lists relates two lists element-wise according to a relation on the elements. This relator is defined inductively: For each relation R, $\langle R\rangle list\_rel$ is the smallest relation that satisfies

$$(Nil,Nil) \in \langle R\rangle list\_rel$$
$$[\![\ (a,a') \in R;\ (l,l') \in \langle R\rangle list\_rel\ ]\!] \Longrightarrow (Cons\ a\ l,\ Cons\ a'\ l') \in \langle R\rangle list\_rel$$

Similarly, natural relators can be defined for other algebraic types. The natural relator $\to$ for functions relates functions that produce related results when applied to related arguments. It is defined as

$$(f,\ f') \in R_a \to R_r \longleftrightarrow \forall (x,x') \in R_a.\ (f\ x,\ f'\ x') \in R_r$$

Functions and algebraic types are usually refined to themselves using their natural relators. However, types like maps or sets need to be represented differently. The relator *list_set_rel*, which relates distinct lists to finite sets, is defined as

$$\langle R\rangle list\_set\_rel = \langle R\rangle list\_rel \circ \{\ (l,s).\ s = set\ l \wedge distinct\ l\ \}$$

Here, $\circ$ is relational composition. That is, a list of concrete elements is first related to a list of distinct abstract elements, and this list is then related to a set of abstract elements.

[2] Relators are typically required to be monotonic, commute with composition and converse, and preserve identity [3]. However, as our technique does not rely on this, we call relator any function with the appropriate type. Actually, most of our relators satisfy these properties, a notable exception being the function relator.

2.3 Transfer Rules

In Isabelle/HOL, a program is represented as a term, which contains no schematic variables. Thus, in order to relate a concrete and an abstract term, we have to relate applications, abstractions, constants and free variables. We assume that an abstract constant (or free variable) $f'::'a_1 \to \ldots \to 'a_n$ is implemented by a concrete constant (or free variable) $f::'c_1 \to \ldots \to 'c_n$ of the same arity. In order to relate these constants, we have to prove a *transfer rule* of the form $(f,f') \in R_1 \to \ldots \to R_n$. For abstraction and application, we use the following transfer rules:

$$[\![\bigwedge x\ x'.\ (x,x') \in R_a \Longrightarrow (t,t') \in R_r]\!] \Longrightarrow (\lambda x.\ t,\ \lambda x'.\ t') \in R_a \to R_r$$
$$[\![(f,f') \in R_a \to R_r;\ (x,x') \in R_a]\!] \Longrightarrow (f\ x,\ f'\ x') \in R_r$$

We now state the *synthesis problem* that our tool has to solve: Given an (abstract) term t' and transfer rules for its constants and free variables, synthesize a (concrete) term t and a relation R, such that $(t,t') \in R$ can be proven by the transfer rules.

Note that the synthesis problem is effective: As the rules decompose the structure of the abstract term, there are only finitely many proof trees for each term t'. In Isabelle/HOL, all solutions to the synthesis problem can be enumerated by solving the goal $(?t,t') \in ?R$ by repeated resolution with the transfer rules, using backtracking to recover from failed attempts or to explore further solutions. However, this approach may produce large search spaces. In Section 3 we describe our actual implementation of the synthesis.

Example 2. The transfer rules for the list constructors (cf. Example 1) are

$$(Nil,Nil) \in \langle R \rangle list\_rel$$
$$(Cons,Cons) \in R \to \langle R \rangle list\_rel \to \langle R \rangle list\_rel$$

Now consider the relator $int\_nat\_rel = \{(i,n).\ i = int\ n\}$ that relates integers to natural numbers.[3] The transfer rule for addition is

$$(op\ +,\ op\ +) \in int\_nat\_rel \to int\_nat\_rel \to int\_nat\_rel$$

Note that in Isabelle/HOL the +-operator is overloaded for both integers and natural numbers.

Moreover, consider the abstract term $t' = \lambda x\ y::nat.\ [x+y]$ that maps two natural numbers to a list, where $[x+y]$ is syntactic sugar for $Cons\ (x+y)\ Nil$. Trying to prove the goal $(?t,t') \in ?R$ by recursive resolution with the transfer rules results in the theorem

$$(\lambda x\ y::int.\ [x+y],\ t') \in int\_nat\_rel \to int\_nat\_rel \to \langle int\_nat\_rel \rangle list\_rel$$

2.4 Equality and Type Classes

Some refinements also depend on operations that are implicit on the abstract type, like equality or type class operations. In this case, the concrete operation

[3] Implementing natural numbers by integers makes sense, as Isabelle/HOL uses a binary representation for integers, but a unary one for natural numbers.

needs to be parameterized by explicit concrete versions of these implicit abstract operations. Then, the transfer rules have the more general form $[\![(c_1, a_1) \in R_1; \ldots; (c_n, a_n) \in R_n]\!] \Longrightarrow (c\ c_1\ \ldots\ c_n, a) \in R$, where the constants a_i are the implicit abstract operations, and c_i are their implementations. Note that the synthesis problem remains effective, as repeated resolution with the transfer rules can only produce finitely many different subgoals. Thus, the finitely many possible proof trees where no subgoal occurs twice on a path can be enumerated.

Example 3. Reconsider the relator *list_set_rel* from Example 1, which implements finite sets by distinct lists. The membership operation $\in$ is implemented by searching the list for an equal element:

primrec *glist_member* :: (*'a*→*'a*→*bool*) → *'a* → *'a list* → *bool* **where**
glist_member eq x [] ⟷ *False*
| *glist_member eq x* (*y*#*ys*) ⟷ *eq x y* ∨ *glist_member eq x ys*

It is parameterized with an equality operation. Note that we cannot use the default equality operation on the concrete side, as equality of abstract values does not necessarily imply equality of their implementations. For example, the set $\{1, 2\}$ is implemented by both lists, $[1, 2]$ and $[2, 1]$.

The transfer rule for the membership operation is the following:

(*eq,op* =) ∈ *R*→*R*→*Id* ⟹ (*glist_member eq,op* ∈) ∈ *R* → ⟨*R*⟩*list_set_rel* → *Id*

Thus, in order to transfer membership, we need to synthesize an additional equality operation on the element type. Note that we relate Booleans by their natural relator *Id*.

Other examples for implicit operations are hash codes and ordering operations, which are usually defined by type classes on the abstract type. Moreover, transfer rules with premises can be used to automatically instantiate generic algorithms, as described in Section 3.5.

2.5 Summary

In this section, we have described the basic machinery required to synthesize a (concrete) term t and a relation R from an (abstract) term t' such that $(t, t') \in R$ holds. In the next section, we tackle the additional problems that arise when using these ideas to implement a tool for automatic transfer of abstract programs to efficiently executable ones.

3 Tool Implementation

Given an abstract program specified by a term t', we ideally want to synthesize a term t and a relation R such that 1: $(t,t') \in R$ holds, 2: t is executable, 3: R is adequate, and 4: t is optimally efficient. Criterion 1 is a by-product of our synthesis that works by actually proving $(t,t') \in R$. In the Isabelle/HOL setting, Criterion 2 has to be understood w. r. t. the code generator [7,9], which exports a

functional fragment of HOL to functional languages like ML or Scala. If the left-hand sides of the transfer rules lay in this functional fragment, the synthesized term is executable. Thus, it is the responsibility of the user to specify transfer rules with executable left-hand sides. Otherwise, exporting the synthesized term will fail. Criterion 3 considers the refinement relations itself. A refinement relation should be able to uniquely represent a sufficiently large subset of the abstract type. Again, it is the responsibility of the user not to use inadequate refinement relations. Otherwise, the synthesized refinement theorem will be too weak to prove useful properties about the synthesized term. Finally, Criterion 4 is the most difficult to achieve, as it depends on many parameters outside the scope of our tool, like the algorithm itself, the expected distribution of input, etc. However, we provide some heuristics for selecting efficient implementations, as well as many configuration options that allow the user to fine-tune the result.

In the remainder of this section, we describe the actual implementation of Autoref and the heuristics.

3.1 Identification of Operations

The first step to solve the synthesis problem is to identify the abstract operations. In the previous section, we optimistically assumed that relators match the structure of the type and operations in the abstract term are expressed by single constants. However, these assumptions are not true in practice. For example, Isabelle/HOL represents maps from *'a* to *'b* by the type $'a \to 'b\ option$. Lookup in a map is expressed by function application, the empty map is $\lambda x.\ None$ and map update is $fun\_upd\ m\ k\ (Some\ v)$[4].

To handle this mismatch, we represent conceptual types like map or set by so called *interfaces*. We write $f :_i I$ to express that operation f has interface I. In order to identify operations that are not represented by a single constant, we define a set of *pattern rewrite rules* of the form $pat \equiv c\ x\_1 \ldots x\_n$.

The actual operation identification is then done by solving a type inference problem according to the following rules:

$$ctxt: \frac{x : I \in \Gamma}{\Gamma \vdash x : I} \quad pat: \frac{t \equiv t' \quad \Gamma \vdash t' : I}{\Gamma \vdash t : I} \quad const: \frac{c :_i I}{\Gamma \vdash c : I}$$

$$app: \frac{\Gamma \vdash t : I_1 \quad \Gamma \vdash f : I_1 \to I_2}{\Gamma \vdash f\ t : I_2} \quad abs: \frac{(x : I_1)\Gamma \vdash t : I_2}{\Gamma \vdash (\lambda x.\ t) : I_1 \to I_2}$$

Here, $I_1 \to I_2$ is the interface for functions. The rules are standard except for the *pat* rule, which replaces the current term according to a pattern rewrite rule. Our type inference algorithm first tries to apply the *pat* rule. Only if this does not lead to a valid typing, it backtracks to use the *const*, *app*, or *abs* rules. If a typing is found, the term is rewritten according to the applied *pat* rules[5].

[4] The forms *Map.empty* and $m(k \mapsto v)$ are just syntactic sugar.

[5] The actual implementation combines type inference and rewriting.

Moreover, to simplify later processing, we define tagging constants OP and \$ to indicate operations and application of operands: $OP\ c = c$ and $f\$\, x = f\, x$.

Example 4. The interface for maps is $\langle I_k,I_v\rangle i\_map$. Note that we use the same postfix notation for arguments of interfaces as for arguments of relators. The pattern rewrite rule for map lookup is $m\ k \equiv op\_map\_lookup\ m\ k$, and we have $op\_map\_lookup :_i \langle I_k,I_v\rangle i\_map \to I_k \to \langle I_v\rangle i\_option$.

Now consider the term $m\ k :: 'a\ option$. Autoref has to decide whether this is a lookup operation in the map m, or an application of the function m. The type inference first tries the pattern rewrite rule $m\ k \equiv op\_map\_lookup\ m\ k$, thus trying to derive a map interface for m. If this fails, it backtracks and uses the *app*-rule to derive a function interface for m. If m really has a map interface, after rewriting and adding the OP and \$ tags, the term becomes $OP\ op\_map\_lookup\,\$\, m\,\$\, k$. Otherwise, it becomes $m\,\$\, k$.

3.2 Selecting the Implementation Types

After it has identified the operations of the abstract term, Autoref decides what concrete types to use. We separate this decision from the actual synthesis mainly for efficiency reasons. In early versions of Autoref, we had serious efficiency problems due to extensive backtracking in the synthesis phase.

The goal of the next phase is to annotate each operation in the abstract term by the relation that will be used to transfer it to a concrete operation. This annotation is done by another tagging constant ::: that is defined as $f ::: R = f$. In order to influence the result of this phase, the user can manually place :::-annotations in the abstract term. This phase also implements some heuristics that aim at choosing efficient implementations.

Internally, this phase is split into multiple sub-phases, which successively instantiate relation variables to actual relations.

The first sub-phase uses the derived interface types to annotate each operation with a relation that consists of fresh variables and function relators, and also processes :::-annotations. After this sub-phase, every operation is annotated with a relation. Typically, most of these relations still contain fresh variables, and only a few have been specified by the user via explicit annotations.

The next sub-phase tries to restrict the possible instantiations of the relation variables by what we call *homogeneity rules*. The idea is that operations should preserve the implementation if possible. For this purpose, there is a set of homogeneity rules of the form $OP\ f ::: R$, and Autoref tries to unify the annotated operations in the term against the homogeneity rules, using a depth-first strategy. For each operation, a maximal specific homogeneity rule that has a unifier is taken. If there is no such rule, the original relation is not changed. This method propagates the user annotations over the operations, according to the homogeneity rules. The depth-first strategy ensures that user annotations are propagated upwards in the term, until they conflict with other user annotations.

Example 5. A typical setup provides a generic implementation for the set intersection operation, which iterates over the first set, performs a membership query

in the second set, and builds up the result set. It may be instantiated for any combination of implementations of the first, second, and result set. Moreover, consider two set implementations with the relators *rbt_set_rel* and *list_set_rel*.

Assume the user wants to translate the term $a \cap (b ::: \langle R \rangle list\_set\_rel)$. After operator identification and relator annotation, the term becomes:

$$(OP \cap ::: ?R_1 \to \langle R \rangle list\_set\_rel \to ?R_2)\ \$\ a\ \$\ b$$

where $?R_1$ and $?R_2$ are fresh relator variables that need to be instantiated further. Autoref could safely use any relation for $?R_1$ and $?R_2$, as the generic implementation of $\cap$ works for any combination of relations. However, the user probably wanted both a, and the result of the intersection to be implemented via *list_set_rel*. This can be expressed by the homogeneity rule $OP \cap ::: R \to R \to R$. If applied, it instantiates both $?R_1$ and $?R_2$ to $\langle R \rangle list\_set\_rel$.

Now assume the user specified $(a \cap b) ::: \langle R \rangle rbt\_set\_rel$, thus explicitly requesting the result to be implemented by *rbt_set_rel*. Again, the homogeneity rule ensures that both a and b are implemented by *rbt_set_rel*, unless they contain different annotations.

Another useful homogeneity rule is $OP \cap ::: R \to R \to R'$, which tries to at least choose the same representation for both operands.

A homogeneity rule should not be able to render a possible implementable operation unimplementable. For example, if the only implementations of operation f' are $(f_1, f') ::: R_1 \to R_2$ and $(f_2, f') ::: R_2 \to R_1$, the homogeneity rule $OP\ f ::: R \to R$ would make the operation unimplementable.

After application of the homogeneity rules, the term may still contain uninstantiated relation variables. In the final sub-phase, all relation variables are instantiated by means of the available transfer rules. For each operation $OP\ f ::: R$ in the term, we try to find a transfer rule with a conclusion $(\_, f) \in R'$ such that R unifies with R', and instantiate R accordingly. This instantiation is done in a depth-first order, using backtracking until a solution is found. Premises of transfer rules are taken into account only if they have the form $(\_,\_) \in \_$.

In order to influence the solution, the transfer rules are ordered by priorities, such that rules with higher priority are tried first.

The priority of a rule is computed from a direct component, which may be annotated to the rule, and a relator component, which prefers transfer rules involving certain relators. For example, in order to prefer red-black trees over lists, one gives the relator *rbt_set_rel* a higher priority than *list_set_rel*. On the other hand, to prefer an optimized implementation of an operation over an unoptimized one, the transfer rule for the optimized implementation is annotated with a higher direct priority.

Note that the relator annotation phase may render solvable synthesis problems unsolvable. One reason are unsuitable homogeneity rules, as described above. Another reason is that the last subphase does not consider all side conditions of transfer rules. However, when carefully setting up homogeneity and transfer rules, those effects will not occur. Thus we chose to accept this incompleteness for the advantage of a considerably faster synthesis.

3.3 Side Conditions

Apart from requiring implementations of equality and other type class operations, transfer rules may have other side conditions that need to be solved. For example, the Refinement Framework [19] sometimes requires relations to be single-valued, and functions to be monotonic. It already provides solvers for those properties, which we invoke from our tool.

Another complication arises for transfer rules with preconditions over operands. For example, the *hd*-operation, which returns the first element of a list, can only be transferred if the list is non-empty. Hence, the transfer rule for *hd* cannot be written in the form $(hd,hd) \in \langle R \rangle list\_rel \to R$. We solve this problem by also allowing transfer rules written in first-order form:

$$[\![\ l' \neq [];\ (l,l') \in \langle R \rangle list\_rel\ ]\!] \Longrightarrow (hd\ l,\ hd\ l') \in R$$

When applying transfer rules in first-order form to operations that do not have enough arguments, the operation is η-expanded. Note that η-expansion is always possible in Isabelle/HOL, as $f = \lambda x.\ f\ x$ is a theorem.

In order to be able to solve the side conditions, we have to augment some transfer rules to pass on additional information. For example, in order to transfer the term $If\ (l \neq [])\ (hd\ l)\ a$, we have to pass on information about the *If* statement during the transfer. For this purpose, we again use a first-order transfer rule:

$$[\![\ (c,c') \in Id;\ c \Longrightarrow (t,t') \in R;\ \neg c \Longrightarrow (e,e') \in R\ ]\!] \Longrightarrow (If\ c\ t\ e,\ If\ c'\ t'\ e') \in R$$

Thus, when transferring the *hd*-operation, $l \neq []$ is available as an assumption. We use similar rules for other crucial operations like the assertions from the Refinement Framework.

Side conditions may also be used for optimized implementations. Consider, for example, the insert operation for a set represented by a distinct list. If we know that the element is not yet contained in the set, it can be implemented in constant time by prepending the element to the list. Otherwise, we need linear time to check whether the element is already contained in the list. By giving the transfer rule for the optimized operation a higher priority, it is tried first. If it's side condition can be solved, the optimized version is used. Otherwise, the synthesis backtracks and uses the general transfer rule.

3.4 Synthesis

The last phase of Autoref takes the term t', which is completely annotated with relations, and constructs a proof goal of the form $(?t,t') \in ?R$. Here, $?t$ is a schematic variable that will be instantiated during the synthesis process, and R is the relation inferred for t' by the previous phase. Then, it tries to apply the transfer rules to this goal in order of their priorities. After applying each transfer rule, the process is recursively invoked for the evolving subgoals. If solving one of the subgoals fails, the next matching transfer rule is tried. If the subgoal is not of the form $(t, t') \in R$, it is a side condition and Autoref analyzes its shape to find an adequate solver. As an additional optimization, the premises of a transfer rule

are ordered such that side conditions concerning the abstract term or the relator come first. This avoids synthesizing the concrete term when side conditions over the abstract term or relation fail.

3.5 Generic Programming

Many abstract operations can be implemented in terms of other abstract operations. For example, we have $a \cap b = \{\} \longleftrightarrow \forall x \in a.\ x \notin b$, i. e. the disjointness test for sets can be implemented by means of bounded quantification and membership query. Along these lines, most operations on finite sets can be implemented by five basic operations: Empty set, insert, membership, deletion of an element, and iteration over the elements of the set. We extensively exploit this idea already in the Isabelle Collection Framework [16,15]. However, there we have to manually pre-instantiate the generic algorithms for each combination of implementations, which does not scale. Using Autoref, generic algorithms are expressed as transfer rules, and automatically instantiated only on demand. Moreover, the usage of generic algorithms is transparent to the user, who specifies an abstract operation, and lets the tool decide whether it is realized by a direct implementation or a generic algorithm.

Example 6. Reconsider the disjointness test. We define the constant

$$op\_set\_disjoint\ a\ b \longleftrightarrow a \cap b = \{\}$$

and add an appropriate rewrite rule to the operation identification phase. Moreover, we define

$$gen\_disjoint\ ballI\ memI\ a\ b = ballI\ a\ (\lambda x.\ \neg memI\ x\ b)$$

Then, we can easily prove the following transfer rule:

$$[\![\ (b, Ball) \in \langle Re \rangle Rs_1 \to (Re \to Id) \to Id;\ (m,\ op\ \in) \in Re \to \langle Re \rangle Rs_2 \to Id\ ]\!] \\ \Longrightarrow (\ gen\_disjoint\ b\ m,\ OP\ op\_set\_disjoint) \in \langle Re \rangle Rs_1 \to \langle Re \rangle Rs_2 \to Id$$

A low direct priority ensures that it does not override explicit rules for disjointness tests. Thus, whenever Autoref finds no explicit rule for a disjointness test, it tries to find rules for bounded quantification and membership instead, and automatically implements the disjointness test by those operations.

Using such rules, we have to be careful not to follow cycles, trying to implement an operation by means of itself. Checking for such cycles is not yet implemented. Thus, it is the responsibility of the user not to use transfer rule setups with cyclic dependencies. However, even with this restriction, we were able to implement generic algorithm libraries for maps and sets (cf. Section 4.3).

3.6 Summary

In this section we have described how to use the basic idea of synthesis via transfer rules to implement the Autoref tool, which automatically synthesizes efficient implementations of abstractly specified algorithms. The tool has several heuristics

that try to automatically produce a suitable implementation. If these heuristics produce a non-adequate result, the user can influence the result by configuration of the heuristics and annotations to the abstract algorithm. In the next section, we present some case studies that prove the practical usefulness of Autoref.

4 Case Studies

In this section, we describe the integration of Autoref with the Isabelle Refinement Framework [18,19] and the Isabelle Collection Framework [16,15]. Moreover, we describe a library of generic map and set algorithms that demonstrates the generic programming capabilities of Autoref. Finally, we report on the automatic refinement of some complex algorithms to efficiently executable code.

4.1 Refinement Framework

In order to be useful for practical algorithms, we have set up Autoref as a back end to the stepwise refinement development process provided by the Isabelle Refinement Framework [18,19].

A detailed description of the Refinement Framework can be found in [19]. Here, we give a very brief overview. The basic concept of the Refinement Framework is a nondeterminism monad, whose inner type is called *result*. A result is either a set of values, describing the possible outcomes of a nondeterministic computation, or it is the special result **fail**, describing that one of the possible outcomes is an exception, i. e. a failed assertion or diverging computation. By lifting the subset ordering, with **fail** being the biggest element, one gets a complete lattice structure on results. The lifted ordering is called *refinement ordering*, where smaller results are more refined. An algorithm is expressed as a function yielding a result. Correctness of an algorithm is expressed by refinement of its specification, e. g. $\Phi \Longrightarrow f\, x \leq \mathbf{spec}\ \Psi$ describes correctness of f w. r. t. precondition Φ and postcondition Ψ. Here, $\mathbf{spec}\ \Psi$ is the result that contains all values satisfying Ψ.

Given a (single-valued) refinement relation R, the concretization function $\Downarrow R$ maps abstract results to concrete results w. r. t. R. Thus, data refinement is expressed by $r \leq \Downarrow R\ r'$, meaning that r refines r' w. r. t. the relation R.

In order to integrate the Refinement Framework with Autoref, we define data refinement as a relator for results: $\langle R\rangle nres\_rel = \{(c,a).\ c \leq \Downarrow R\ a\}$. Then, we provide transfer rules for the combinators of the Refinement Framework. Those transfer rules are already contained in the Refinement Framework, and only have to be rephrased in the format expected by Autoref. Some of the transfer rules have side conditions, for which the Refinement Framework already provides solvers, which could easily be integrated into Autoref.

4.2 Collection Framework

The Isabelle Collection Framework [15,16] provides a rich library of verified collection data structures, and is already based on data refinement. Thus, it is

straightforward to set up Autoref to use the data structures provided by the Collection Framework.

However, the Collection Framework only supports refinement relations of the form $\langle Id,\ldots,Id\rangle R$. For example, it is not possible to refine a set of sets of integers to a list of lists of integers. Thus, we implemented a red-black tree based map implementation and a list-based set implementation that do not have this restriction. Using parametricity [26], we were able to reuse the existing theorems about red-black trees and lists, as illustrated in the following example:

Example 7. The existing implementation of sets by distinct lists gives us the following transfer rule:

$$(list\_member, Set.member) \in Id \to \{(l,s).\ s = set\ l \wedge distinct\ l\} \to Id$$

Here, *list_member* implicitly uses equality on the elements. It is straightforward to show $list\_member = glist\_member\ (op =)$, where *glist_member* is the one from Example 3. Moreover, Autoref easily shows that *glist_member* is parametric[6]:

$$(glist\_member,\ glist\_member) \in (R \to R \to Id) \to R \to \langle R\rangle list\_rel \to Id$$

Combining these theorems, one gets precisely the transfer rule from Example 3.

4.3 Generic Programming

In Section 3.5, we sketched how Autoref can be used for generic programming. In order to demonstrate this feature, we implemented a library of generic map algorithms, which provides a variety of operations based on the five basic operations empty, update, lookup, delete, and iterate. Analogously, we implemented generic set algorithms based on the basic operations empty, insert, member, delete, and iterate. Finally, we implemented the basic set operations by the basic map operations, using a map from elements to unit values to represent a set.

Thus, in order to prototype a new data structure, it is enough to implement the five basic map operations. All other map and set operations become available automatically. Most of the generic algorithms are reasonable efficient, such that they can be kept even for the final version. To specialize a generic algorithm for a particular implementation, it is sufficient to add the specialized transfer rule with a higher priority than the generic rule. For example, we have a generic algorithm for union of finite sets that iterates over one set and inserts its elements into the other set. However, for red-black trees, there is a more efficient algorithm. It is declared as a transfer rule with default priority, thus overriding the lower priority rule for the generic algorithm.

4.4 Code Generation for Actual Algorithms

We have tested Autoref on several actual algorithms. The most complex ones are the algorithm by Ilie, Navarro and Yu for the computation of simulation

[6] Indeed, this is a theorem that you get for free in the setting of [26]!

preorders in nondeterministic finite automata [14], and an emptiness check for Buchi automata using a nested depth-first search [11]. For the former algorithm, we adapted an existing formalization [5], where the refinement to executable code was done manually[7]. Here, the size of the Isabelle text required for the refinement to executable code was reduced from more than 500 lines to about 15 lines. In order to use Autoref, we had to insert two additional assertions into the abstract algorithm, which were required to automatically discharge side conditions of transfer rules. In the original formalization, these side conditions were discharged using some non-trivial reasoning during the manual refinement.

For the latter algorithm, the refinement to executable code requires about 10 lines. Moreover, we require about 20 lines for setup of a custom datatype, for which automation is not yet supported. As this algorithm was initially developed using Autoref, we have no data how big a manual refinement would be, but we estimate it to several hundred lines of code.

4.5 Data Refinement within the Code Generator

The Isabelle/HOL code generator also supports automatic data refinement [8]. However, it has some limitations that render it unsuitable for our purpose, namely code generation for programs defined in the Refinement Framework. For example, the refinement relations are restricted to the form $\alpha\ c = a$. This is essential for integration into the Isabelle/HOL code generator. However, it is not possible to express reduction of nondeterminism, which is required to be used as back end for the Refinement Framework. Moreover, it lacks the operation identification of our tool, thus limiting the refinement to types with their own type constructor. On the other hand, due to the direct integration into the code generator, one gets support of the Isabelle packages for defining recursive functions and algebraic datatypes for free, and tools like evaluate and quickcheck [4] immediately profit from the more efficient code. Here, Autoref currently requires manual setup for each non-primitive recursion scheme and for each algebraic datatype, and automating this task would require quite some effort.

The code generator of Myreen et al. [22] for the HOL4 theorem prover translates terms to the deeply embedded MiniML language, and proves correctness of the translation. It uses a synthesis procedure that is similar to ours, i. e. it keeps track of a relation between the generated code and the original term. While the currently implemented features seems to be limited[8], in theory it should be possible to support the same generality as Autoref does, which yields an interesting topic for future research.

5 Conclusions

We have presented Autoref, a tool for automatic data refinement in Isabelle/HOL. Given an abstract algorithm that uses abstract concepts like maps or sets, it synthesizes a concrete algorithm that uses efficient implementations like red-black

[7] A very early prototype of Autoref was already used for some simple steps.
[8] For example, equality on abstract values is mapped to equality in the target language.

trees, and a corresponding refinement theorem. Autoref allows for both rapid prototyping of executable code and fine-tuning the results to get the final version. Compared to previous manual approaches, our tool saves the user from tedious and time consuming writing of boilerplate code. To substantiate the usefulness of Autoref, we have shown how it can be used to refine actual algorithms for simulation preorder computation and for nested depth-first search.

Directions of future work include to add even more automation. For example, transfer rules with natural relators correspond to the "theorems for free" of [26], and could be derived automatically. Moreover, we are currently working on several algorithm verifications using the Refinement Framework. The feedback from those projects will lead to improvements of the tool, and extension of its data structure and generic algorithm libraries. Another interesting topic is to use the heuristics that we developed for Autoref as a front end to the code generator based data refinement [8].

Acknowledgements. We thank Andrei Popescu for proofreading and inspiring discussions about parametricity, and the anonymous reviewers for their useful comments.

References

1. Back, R.J.: On the correctness of refinement steps in program development. Ph.D. thesis, Department of Computer Science, University of Helsinki (1978)
2. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
3. Backhouse, R.C., de Bruin, P., Malcolm, G., Voermans, E., van der Woude, J.: Relational catamorphisms. In: Proc. of the IFIP TC2/WG2.1 Working Conference on Constructing Programs. Elsevier Science Publishers BV (1991)
4. Bulwahn, L.: The new quickcheck in Isabelle: Random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer, Heidelberg (2012)
5. Eberl, M.: Efficient and Verified Computation of Simulation Relations on NFAs. Bachelor's thesis, Technische Universität München (2012)
6. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. To appear in Proc. of CAV (2013)
7. Haftmann, F.: Code Generation from Specifications in Higher Order Logic. Ph.D. thesis, Technische Universität München (2009)
8. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer, Heidelberg (2013)
9. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
10. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1, 271–281 (1972)
11. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Proc. of SPIN Workshop. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. American Mathematical Society (1997)

12. 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)
13. Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: Isabelle Users Workshop 2012 (2012)
14. Ilie, L., Navarro, G., Yu, S.: On NFA reductions. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds.) Theory Is Forever. LNCS, vol. 3113, pp. 112–124. Springer, Heidelberg (2004)
15. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010)
16. Lammich, P.: Collections framework. In: Archive of Formal Proofs, formal proof development (December 2009), `http://afp.sf.net/entries/Collections.shtml`
17. Lammich, P.: Tree automata. In: Archive of Formal Proofs, formal proof development (December 2009), `http://afp.sf.net/entries/Tree-Automata.shtml`
18. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs, formal proof development (2012), `http://afp.sf.net/entries/Refine_Monadic.shtml`
19. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012)
20. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 216–232. Springer, Heidelberg (2011)
21. Musser, D.R., Stepanov, A.A.: Generic programming. In: Gianni, P. (ed.) ISSAC 1988. LNCS, vol. 358, pp. 13–25. Springer, Heidelberg (1989)
22. Myreen, M.O., Owens, S.: Proof-producing synthesis of ML from higher-order logic. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, pp. 115–126. ACM (2012)
23. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
24. Nordhoff, B., Lammich, P.: Formalization of Dijkstra's algorithm, formal proof development (2012)
25. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress, pp. 513–523 (1983)
26. Wadler, P.: Theorems for free! In: Proc. of FPCA, pp. 347–359. ACM (1989)

Data Refinement in Isabelle/HOL

Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow

Technische Universität München

Abstract. The paper shows how the code generator of Isabelle/HOL supports data refinement, i.e., providing efficient code for operations on abstract types, e.g., sets or numbers. This allows all tools that employ code generation, e.g., Quickcheck or proof by evaluation, to compute with these abstract types. At the core is an extension of the code generator to deal with data type invariants. In order to automate the process of setting up specific data refinements, two packages for transferring definitions and theorems between types are exploited.

1 Introduction

Algorithm verification is most convenient at a high level of abstraction, reasoning about data in terms of sets, functions and other mathematical concepts. However, when running the verified code we want to replace sets and functions by lists and trees, to make them efficiently executable. This replacement is called *data refinement*, and the ideal theorem prover should do this fully automatically once we prove that the concrete representation is adequate.

This paper describes a data refinement framework for Isabelle/HOL that automatically replaces abstract data structures by concrete ones during code generation. Our main contribution is a lightweight infrastructure and methodology that reduces data refinement entirely to code generation, requires zero effort from the user and is based on a minimal extension of the code generator.

More formally, data refinement replaces an *abstract* data type A by a more *concrete* one C in the generated code. The typical example is the implementation of sets by lists. The concrete type is also called the *implementation* or *representation*. Refining A by C requires an *abstraction function* $Abs :: C \to A$ (e.g., mapping $[1, 2]$ to $\{1, 2\}$) and an *invariant* $inv :: C \to bool$ (e.g., ruling out lists with duplicates). The basic picture is shown in Figure 1.

The standard approach is to demand that Abs is a homomorphism: for every operation $f \in \Sigma$ (the primitive operations that need implementing) on the abstract type and its concrete implementation f' it must be shown that $f(Abs(x)) = Abs(f'(x))$. A system supporting data refinement on this basis will require the user to prove the homomorphism property for all operations to ensure soundness of the refinement step. This means that a new trusted component is added to the system, the refinement manager. A typical example for this approach is the KIV system [18].

We turn the approach on its head: Rather than check that the correct homomorphism theorems have been proved before code is generated, the homomorphism theorems themselves are the glue code between f and f'. More precisely,

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 100–115, 2013.

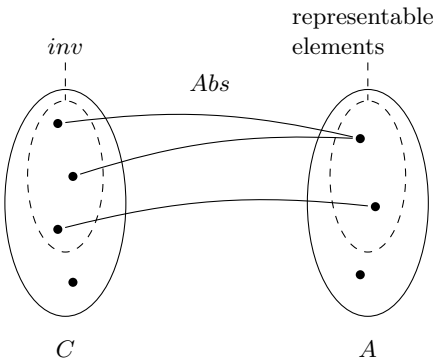


Fig. 1. Data refinement

we instruct the code generator to view A as an algebraic data type with the single (uninterpreted!) constructor $Abs :: C \to A$. Now $f(Abs(x)) = Abs(f'(x))$ is a code equation that performs pattern matching on Abs to turn a call of f into a call of f'. This is the key point of our approach: We generate code for the actual function f, not for some other function f' for which some additional theorems show that it implements f in the correct manner. This form of data refinement is completely automatic: Once a particular refinement of A by C has been set up, generating code involving functions on A involves no further input by the user. This works amazingly well and is explained in §2.

Unfortunately it breaks down once we have a non-trivial invariant and can only prove $inv(c) \implies f(Abs(c)) = Abs(f'(c))$. This is a conditional equation and thus unsuitable for generating code. At this point we need to introduce a minimal extension of the code generator that deals with invariants. This is the contents of §3, where the correctness of the extension is also proved.

As a final generalization we allow A to be a nested type expression. This complicates matters and is the subject of §4.

Data refinement is crucial for Isabelle/HOL because it enables code generation for some of the most important types, namely sets and numbers. The details follow, but we can already mention that this is essential for two important applications, in addition to explicit algorithm development by the user: Quickcheck, Isabelle's automatic counterexample search facility [2], and proof by evaluation. Both take advantage of default implementations of sets and numbers to execute seemingly abstract statements like $\{1, 1/2\} \cap \{1 - 1/2, 2\} = \{1/2\}$.

1.1 Code Generation

Isabelle/HOL supports code generation for a number of functional programming languages (SML, OCaml, Haskell, Scala). Basically, equational theorems in HOL, called *code equations*, are translated into function definitions in the target languages. A mathematical treatment of this translation process, including correctness proofs, can be found elsewhere [5]. We stay on the level of code

equations here and do not need to worry about the further translation steps. The key correctness property of the generated code is that any evaluation in the target language corresponds to an equality provable in HOL. In other words, the generated code is a fast rewrite engine, which can be used to derive equations.

When we want to generate code for some function f, any list of equations of the form $f\ \ldots = \ldots$ (with pattern matching on the lhs) can (in principle) serve as code equations, not just the original definition of f. Thus we are free to define a second, more efficient function g, prove $f(x) = g(x)$, and use this equation (together with the ones for g) as the code equations for f.

Algebraic data types in HOL are turned into equivalent algebraic data types in the target language. Interestingly, the correctness proof revealed that in fact any function in HOL can in principle become a constructor function in the target language (but of course not a defined function at the same time).

1.2 Isabelle/HOL

Isabelle/HOL [17] is based on Church's simple type theory. Types τ are built from type variables (denoted by α, β, ...) and type constructors κ with a fixed arity. The function type is $\to$ as usual. The notation $t :: \tau$ means that term t has type τ. In concrete examples we use Isabelle/HOL's syntax: $\Rightarrow$ instead of $\to$ and 'a instead of α. The qualified name $A.f$ refers to function f from theory A. Besides $\forall$, we also use this symbol $\bigwedge$ for universal quantification.

In our examples we employ the usual standard types: lists ('a list), sets ('a set), booleans (bool), and the type of optional values ('a option) with constructors Some and None. The primitive way of introducing new types in Isabelle/HOL is the **typedef** command. It takes a non-empty set $S :: \tau\ set$, defines a new type σ, and axiomatizes two isomorphisms $Abs :: \tau \to \sigma$ and $rep :: \sigma \to \tau$ as follows:

$$\forall x.\ rep\ x \in S \tag{1}$$

$$\forall x.\ Abs\ (rep\ x) = x \tag{2}$$

$$\forall x \in S.\ rep\ (Abs\ x) = x \tag{3}$$

Thus the axiomatization requires that the image of rep is in S, Abs is a left-inverse for rep and rep is a left-inverse for Abs on S.

We use another axiomatization of Abs and rep in our paper:

$$\forall x.\ Abs\ (rep\ x) = x \tag{4}$$

$$\forall x.\ x \in S \longleftrightarrow rep\ (Abs\ x) = x \tag{5}$$

The axiomatizations are equivalent [1]: (4) is the same as (2) and the left-to-right direction of (5) is (3). The right-to-left direction of (5) can be derived from (1) by substituting $Abs\ x$ for x. On the other hand, (1) can be derived from (5) by substituting $rep\ x$ for x and using (4). If S is given by a set comprehension $\{x :: \tau.\ P\ x\}$, we often write $P\ x$ instead of $x \in S$.

[1] We want to thank one of the reviewers that found a bug in Nitpick, which found an incorrect counterexample that appeared to show that the two axiomatizations are not equivalent.

2 Basic Data Refinement

We start by considering the situation where there is no invariant. The standard example is the implementation of sets by lists with no restrictions on the order or multiplicity of the elements in the lists. More efficient representations are considered later on, but this one illustrates the basic method well.

The relation between lists and sets is an instance of Figure 1 where $C =$ 'a list, $A =$ 'a set, inv is true everywhere (every list is a valid representation) and $Abs =$ set, a predefined function that returns the set of elements in a list. Infinite sets are not representable (but see the end of the section).

As explained in the Introduction, for code generation purposes we will now consider the abstraction function set as the single constructor of type 'a set. Arbitrary constants (of an appropriate type) can be turned into data type constructors in the generated code (see §1.1). We call such constants *pseudo-constructors*. Like ordinary constructors, they have no defining code equations but other code equations can use them in patterns on the left-hand side. There are no particular logical properties that such pseudo-constructors have to satisfy—they do not have to be injective or exhaust the abstract type. This is how we instruct the code generator to view the function set :: 'a list ⇒ 'a set as a constructor:

code_datatype set

Thus in the generated code the type 'a set will become a data type whose elements are in fact lists, but wrapped up in the constructor set. For the primitive set operations we can easily prove alternative equations that pattern-match on set. Here are some examples:

lemma [code]: {} = set []
lemma [code]: Set.insert x (set xs) = set (List.insert x xs)
lemma [code]: Set.remove x (set xs) = set (List.removeAll x xs)

The [code] tag tells the code generator that a theorem should be considered a code equation and used instead of the original definition of the function involved.

The technique allows the replacement of one type by another type with surprising ease, based purely on the equational semantics of the code generator.

We now generalize from the example. We assume that $A = (\overline{\alpha})\kappa$, where κ is a type constructor and $\overline{\alpha}$ a list of type variables, its arguments; C is unrestricted. We start by defining $Abs :: C \to A$ and registering it as a pseudo-constructor (via **code_datatype**) in order to pattern-match on it in the code equations for the $f \in \Sigma$. There are no restrictions *per se* on the type of f, but in order to abstract the standard pattern seen in the set/list example we need to make some assumptions on the argument types.

Definition 1. *We call a type* $\tau_1 \to \cdots \to \tau_n \to \tau$ basic *(where* τ *is not a function type) iff all* τ_i *are either of the form* $(\ldots)\kappa$ *or do not contain* κ*.*

We assume that all functions in Σ have a basic type, a property that is satisfied by all our applications. Derived functions can of course have arbitrary types.

Now we must prove for each $f \in \Sigma$ a code equation

$$f\ a_1\ \ldots\ a_n\ =\ t$$

where $a_i = Abs(x_i)$ (if $\tau_i = (\ldots)\kappa$) or $a_i = x_i$ (otherwise). The free variables of t must be contained in $\{x_1, \ldots, x_n\}$.

Now terms involving type κ can be handled by the code generator: the code for all primitive functions f has just been proved, and code for derived functions is generated as always. Hence it must be stressed that the only work we need to do is to prove the code equations for the $f \in \Sigma$.

As described above, all occurrences of type κ are refined by the same type. However, our infrastructure does not by itself enforce this: Lochbihler [13] generalizes our approach to multiple representations. He exploits the fact that there can be multiple pseudo-constructors for any type. In fact, Isabelle's default refinement of sets supports cofinite sets, too, by means of a second pseudo-constructor coset :: 'a list ⇒ 'a set where coset $xs = -$ set xs ("$-$" is complement).

3 Data Refinement with Invariants

3.1 Motivation and Example

Implementing sets by lists with possibly repeated elements, as in the previous section, is inefficient. Therefore we now impose the invariant that all elements of the representing lists are distinct and call such lists *distinct lists*. The situation is again the one in Figure 1 with $C =$ 'a list, $A =$ 'a set, $Abs =$ set, but now $inv =$ distinct, a predefined function that tests if all elements of a list are distinct.

But now there is the problem that our pseudo-constructor set can also be applied to lists that are not distinct. As a consequence, some equations for the primitive set operations only hold conditionally, for example

distinct $xs \Longrightarrow$ Set.remove x (set xs) = set (List.remove1 x xs)

This conditional theorem will be rejected as a code equation by the code generator. For soundness reasons the precondition cannot simply be dropped, but without it the theorem does not hold because List.remove1 removes at most one occurrence of x from xs and not all of them like List.removeAll. Our solution is to introduce an intermediate type 'a dlist for distinct lists (see Figure 2). Thus we split the implementation into two steps: the new *subtype* step from 'a list to 'a dlist, where 'a dlist is a new type that is isomorphic to a subset of 'a list, the distinct lists, followed by the basic data refinement of 'a set by 'a dlist which does not involve an invariant anymore and can be dealt with by the method of the previous section.

The new subtype with an invariant is defined by **typedef** (see §1.2):

typedef 'a dlist = {xs::'a list. distinct xs}
  morphisms list Dlist

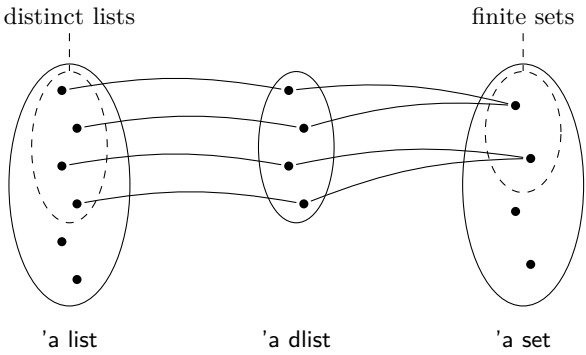


Fig. 2. Sets by distinct lists using 'a dlist

The **morphisms** directive just renames the canonical *rep* and *Abs* functions to

list :: 'a dlist ⇒ 'a list
Dlist :: 'a list ⇒ 'a dlist

As presented in §1.2, we can axiomatize *rep* and *Abs* as follows:

$$\mathsf{Dlist}\ (\mathsf{list}\ dxs) = dxs \tag{6}$$

$$\mathsf{distinct}\ xs \longleftrightarrow \mathsf{list}\ (\mathsf{Dlist}\ xs) = xs, \tag{7}$$

Using the two isomorphisms we can define all primitive operations on dlist by lifting corresponding operations on list. For example, this is the definition of Dlist.remove :: 'a ⇒ 'a dlist ⇒ 'a dlist:

$$\mathsf{Dlist.remove}\ x\ dxs = \mathsf{Dlist}\ (\mathsf{List.remove1}\ x\ (\mathsf{list}\ dxs)) \tag{8}$$

Then we bridge the gap between 'a set and 'a dlist by a new pseudo-constructor dset :: 'a dlist ⇒ 'a set:

$$\mathsf{dset}\ dxs = \mathsf{set}\ (\mathsf{list}\ dxs)$$

If we assume that we already have all primitive operations on the type 'a dlist together with the necessary properties, it is again straightforward to prove code equations implementing set operations, for example for Set.remove:

lemma [code]: $\mathsf{Set.remove}\ x\ (\mathsf{dset}\ xs) = \mathsf{dset}\ (\mathsf{Dlist.remove}\ x\ xs)$

Therefore we turn to the problem of how to implement dlist operations by list operations. Using Dlist as a pseudo-constructor as in the previous section runs into the same problem as before:

$$\mathsf{distinct}\ xs \Longrightarrow \mathsf{Dlist.remove}\ x\ (\mathsf{Dlist}\ xs) = \mathsf{Dlist}\ (\mathsf{List.remove1}\ x\ xs) \tag{9}$$

is only provable under the assumption distinct xs. Therefore we try the definition of Dlist.remove (8) itself as a code equation. Now we need to execute list on the rhs and face the same problem:

$$\mathsf{list}\ (\mathsf{Dlist}\ xs) = xs \tag{10}$$

is only provable if distinct xs. Therefore we extend the code generator for this special case as follows. Attaching attribute [code abstype] to property (6)

lemma [code abstype]: $\mathsf{Dlist}\ (\mathsf{list}\ dxs) = dxs$

instructs the code generator to make Dlist a pseudo-constructor and to turn the composition around and make (10) a code equation, although it is not a theorem. The justification is a meta-theoretic one: we ensure that in code equations, Dlist is only applied to distinct lists, for which (10) is provable. This property of Dlist will be guaranteed by a check that Dlist is only applied to the result of operations on lists that have been proved to preserve the invariant. That is, we ensure that the implementations of the dlist operations on lists preserve distinct. For List.remove1, the implementation of Dlist.remove, we need to show

$$\mathsf{distinct}\ xs \Longrightarrow \mathsf{distinct}\ (\mathsf{List.remove1}\ x\ xs) \tag{11}$$

However, we can do better and combine (8) and (11) like this:

lemma list_remove[code abstract]:
$\mathsf{list}\ (\mathsf{Dlist.remove}\ x\ dxs) = \mathsf{List.remove1}\ x\ (\mathsf{list}\ dxs)$

The attribute [code abstract] instructs the code generator to derive the actual code equation (8) from it (this is a direct consequence of (6)). The lemma also entails (11): if distinct xs then

$$\begin{aligned}&\mathsf{list}\ (\mathsf{Dlist}\ (\mathsf{List.remove1}\ x\ xs)) = \mathsf{list}\ (\mathsf{Dlist.remove}\ x\ (\mathsf{Dlist}\ xs))\\ &= \mathsf{List.remove1}\ x\ (\mathsf{list}\ (\mathsf{Dlist}\ xs)) = \mathsf{List.remove1}\ x\ xs,\end{aligned}$$

i.e., distinct (List.remove1 x xs) (by (9), list_remove, (7)). Thus lemma list_remove also certifies that distinct is preserved by List.remove1.

This concludes the presentation of code generation for dlist. The advantage of our approach is that we have relaxed the principle to only ever generate code from theorems in only one place, equation (10). Above we sketched why this is admissible. In the next subsection we explain our approach in its general form and give a formal correctness proof.

3.2 Subtype Step: The General Case

Now we look at the general form of the subtype step from 'a list (now C) to 'a dlist (now $A = (\overline{\alpha})\kappa$). We have functions $Abs : C \to A$ (Dlist) and $rep : A \to C$ (list) such that $Abs(rep(y)) = y$ and $inv : C \to bool$ (distinct) such that $inv(x) \longleftrightarrow rep(Abs(x)) = x$. We assume that the result type of all functions in Σ contains κ at most at the very outside, e.g., 'a dlist is allowed but 'a dlist list

is not. We discuss this restriction (which does not apply to derived functions) at the end of this section. The format for the code equations is now

$$\psi(f\ \overline{y})\ =\ t$$

where ψ is *rep* (if $\tau = (\ldots)\kappa$) or the identity (otherwise). The free variables of t must be contained in $\overline{y}$. The code generator turns this into $f\ \overline{y} = \phi(t)$ (by a proof step), where ϕ is *Abs* (if $\tau = (\ldots)\kappa$) or the identity (otherwise). The only liberty that the code generator takes is that it turns the theorem $Abs(rep\ y) = y$ into the non-theorem $rep(Abs\ x) = x$. Of course the latter is implied by $inv(x)$, and we will show that $inv(s)$ holds for all terms $Abs(s)$ that may arise during a computation. But this requires a careful proof (see below). The following table summarizes the behavior of the code generator.

E	E'
$rep(f\ \overline{y}) = t$	$f\ \overline{y} = Abs(t)$
$Abs(rep\ y) = y$	$rep(Abs\ x) = x$

Let E be the set of all code equations at the point when the code generator is invoked and let E' be the result of the translation shown in the table above. That is, most equations are moved from E to E' unchanged, but $rep(f\ \overline{y}) = t$ and $Abs(rep\ y) = y$ are translated as above. Moreover, the code generator enforces that *Abs* must not occur on the rhs of any equation in E. (This is not a restriction because if one really needed an operation that behaved like *Abs* one could define it separately from *Abs* to avoid confusion.)

In [5] correctness of the code generation process is shown by interpreting code equations as higher-order rewrite rules and proving that code generation preserves the reduction behavior. Our translation from E to E' is a first step that happens before the steps considered in [5]. We will now prove correctness of that first step by relating the equational theory of E (written $E \vdash u = v$) with reduction in E'. Notation $E' \vdash u \to v$ means that there is a rewrite step from u to v using either a rule from E' or β-reduction.

We call a term t *invariant* iff (i) $E \vdash rep(Abs\ s) = s$ for all subterms $(Abs\ s)$ of t and (ii) every occurrence of *Abs* in t is applied to an argument.

Lemma 1. *If u is invariant and $E' \vdash u \to^* v$, then v is invariant.*

Proof. By induction on the length of the reduction sequence. In each step, we need to check invariance of newly created *Abs* terms. Because user-provided code equations with *Abs* on the rhs are forbidden, only the derived code equation $f\ \overline{y} = Abs(t)$ can introduce a new *Abs* term, namely $Abs(t)$ itself, where *Abs* is applied and for which we have $E \vdash rep(Abs(t)) = rep(Abs(rep(f\ \overline{y}))) = rep(f\ \overline{y}) = t$. Invariance is preserved by β-reduction because it cannot create new *Abs* terms because all *Abs* must already be applied to arguments.

Lemma 2. *If u is invariant and $E' \vdash u \to^* v$, then $E \vdash u = v$.*

Proof. By induction on the length of the reduction sequence. In each step, either β-reduction, or an equation from E, or $f\ \overline{y} = Abs(t)$ (which is a consequence of

E), or $rep(Abs\ x) = x$ is used. Only the last case needs special consideration. By the previous lemma, the subterm $Abs(t)$ of the lhs of the reduction $rep(Abs(t)) \to t$ is invariant, and hence $E \vdash rep(Abs(t)) = t$.

Thus we know that if we start with an invariant term, reduction with E' only produces equations that are already provable in E. Invariance of the initial term is enforced by Isabelle very easily: the initial term must not contain *Abs*.

We have already mentioned that we cannot register a code equation for a basic operation by [code abstract] if the abstract type κ occurs inside the result type rather than at the top level. A workaround is to introduce for each such result type a new abstract type with appropriate projection functions. For example, $\cdots \to (\alpha)\kappa \times (\alpha)\kappa$ becomes $\cdots \to (\alpha)\kappa'$, where κ' is a new abstract type, a copy of $(\alpha)\kappa \times (\alpha)\kappa$ with two projection functions of type $(\alpha)\kappa' \to (\alpha)\kappa$. This leads to simultaneous refinement, which is covered by our approach. Sometimes the workaround can be avoided because the offending operation can be split up into different functions. For example, a function of type $\cdots \to (\alpha)\kappa \times (\alpha)\kappa$ is replaced by two separate functions of type $\cdots \to (\alpha)\kappa$. We believe that the limitation on the result type can be lifted, but it requires a generalization of the correctness proof by employing map functions for each container type involved.

3.3 Using Lifting/Transfer

Building a theory library that implements a new abstract type like 'a dlist can take a bit of work. The main reason is that the type system requires us to convert between values of the concrete and the abstract type with the isomorphisms. This happens in all definitions, for example of Dlist.remove (8). For more complicated types involving higher-order types or other type constructors, more complex combinations of the isomorphisms are required. Then we need to prove the code equations for [code abstract], e.g., lemma list_remove, from those definitions. And finally we need to transfer properties from the concrete to the abstract type. Thus the manual construction of such an abstract type is at least tedious. If one is unfamiliar with the details of the type definition facility, it is not just tedious but cryptic. Hence the success of our approach to invariants depends on the amount of automation we are able to provide for this task.

To automate the construction of abstract types we use the Lifting and Transfer packages [8], which were implemented as general tools but also with the motivation of data refinement in mind. These tools provide automation for building abstract types (subtypes and quotients) in Isabelle/HOL and were inspired by [10]. The Lifting package defines new constants on the abstract level, which is done by *lifting* terms from the concrete level to the abstract level, and proves *transfer rules* relating a term on the concrete level and the newly defined constant. The Transfer package helps to prove theorems on the abstract level (mainly properties of the lifted constants), which is done by *transferring* the goals on the abstract level to goals on the concrete level by using the provided transfer rules.

How to use Lifting/Transfer for implementation of a data structure with an invariant? First of all, we have to set up the lifting infrastructure, which is done by a theorem generated by **typedef** for 'a dlist:

setup_lifting type_definition_dlist

This canonical boilerplate command already registers 'a dlist as an abstract datatype with a constructor Dlist (via [code abstype]).

Then each operation can be lifted by **lift_definition** command:

lift_definition remove :: 'a ⇒ 'a dlist ⇒ 'a dlist **is** List.remove1

This command opens a proof environment with the following goal:

$\bigwedge a\ list.$ distinct $list \Longrightarrow$ distinct (List.remove1 $a\ list$)

The goal merely expresses that the operation on the concrete level preserves the invariant. After the proof is finished, a new constant remove is automatically defined via the correct combination of isomorphisms and also the corresponding code equation list_remove is proved (from the definition and the above goal) and registered in the code generator via [code abstract].

Transfer helps us to prove properties of operations on 'a dlist, in particular code equations:

lemma [code]: Set.remove x (dset dxs) = dset (Dlist.remove x dxs)
apply transfer

Command transfer turns the goal into

$\bigwedge x\ dxs.$ distinct $dxs \Longrightarrow$ Set.remove x (set dxs) = set (List.remove1 x dxs),

which talks about lists rather than distinct lists and can thus be proved easily. Note that dxs is now universally quantified and has type 'a list.

A detailed explanation of the Lifting and Transfer packages is beyond the scope of this paper and full details are going to be published in a forthcoming paper. Here provided description of the Lifting package abstracts from our concrete example of 'a dlist to give a glimpse how the package works internally.

Let us assume we have two morphisms $Abs :: C \Rightarrow A$, $rep :: A \Rightarrow C$ and an invariant $inv :: C \Rightarrow$ bool and we want to lift a function $f :: C \Rightarrow (C)\,\vartheta \Rightarrow C$ to a function f' whose type should be $A \Rightarrow (A)\,\vartheta \Rightarrow A$, where ϑ is a non-abstract type.[2] Then the Lifting package will ask us to prove the correctness condition

$$inv\ x \Longrightarrow \mathrm{pred}_\vartheta\ inv\ y \Longrightarrow inv\ (f\ x\ y),$$

where $\mathrm{pred}_\vartheta :: (\alpha \Rightarrow \mathrm{bool}) \Rightarrow (\alpha)\,\vartheta \Rightarrow \mathrm{bool}$ is a "predicator" for type ϑ extending the predicate inv operating on α's to the predicate $\mathrm{pred}_\vartheta\ inv$ operating on $(\alpha)\,\vartheta$. If we prove the correctness condition, the Lifting package will produce a definition of f' that is equivalent to

$$f'\ x\ y = Abs\ (f\ (rep\ x)\ (\mathrm{map}_\vartheta\ rep\ y)),$$

[2] Our example does not cover the most general type that f could have but it already covers the key concepts: an abstract type, the function type and an abstract type inside of a "container type" (in our example ϑ).

where map_ϑ is a map function for type ϑ. The following code equation is generated and registered in the code generator:

$$rep\,(f'\ x\ y) = f\,(rep\ x)\,(\mathrm{map}_\vartheta\ rep\ y)$$

Finally, a transfer rule relating f and f' is derived and registered in the Transfer package.

In general, if corresponding morphisms, map functions and "predicators" are known to the system, the Lifting package will produce an expected definition for any combination of abstract and concrete types including higher-order and nested types. Except for proving that an operation on the concrete level preserves the invariant, and this is in general unavoidable, everything is fully automatic.

4 From Type Constructors to Type Expressions

4.1 Motivation and Example

The limitation of the code generator is that the type that is being refined has to be of the form $(\overline{\alpha})\,\kappa$. The type of *maps* 'a ⇒ 'b option does not have this form, yet one would still like to refine it by some efficient type of tables. Because 'a ⇒ 'b option is not a plain type constructor, a new type ('a, 'b) mapping has to be introduced. This type is merely a copy of 'a ⇒ 'b option for code generation purposes. It can be refined further, for example, by red-black trees using the techniques from §2 and §3. See Figure 3 for the complete picture. The implementation type rbt-impl is just a plain datatype of binary trees with a color in each node; on top of it the subtype rbt of well-shaped trees satisfying the invariant of red-black trees is defined. This example represents the most general form of data refinement discussed in this paper.

But now all definitions using 'a ⇒ 'b option must be lifted to ('a, 'b) mapping. Of course, it has to be done for primitive operations on maps like a lookup or an update only once for all. But it also has to be done for all other definitions using these primitive functions. The reason is that one has to provide for such derived operations new code equations that use primitive operations of ('a, 'b) mapping and not 'a ⇒ 'b option. On the other hand, no code equations have to be provided for the primitive operations on ('a, 'b) mapping in this phase because these will be provided later on in the phase described in §2 and §3. Of course, it is also possible to base the formalization on the lifted type ('a,'b) mapping from the beginning but this contradicts the very idea of data refinement.

The complications of this general setting are as follows. For a start, you do not obtain code for f but for some f'. This means in particular that none of the tools that build on code generation, e.g., Quickcheck profit from such refinements. Moreover you have to refine every function f to some f', not just the primitive ones, and you have to look carefully at the definition of f' that **lift_definition** actually produced and at the abstraction relations involved to convince yourself that f and f' are in the desired relationship. But it is not quite as bad as this. As soon as you define a derived function h where 'a ⇒ 'b option is no longer

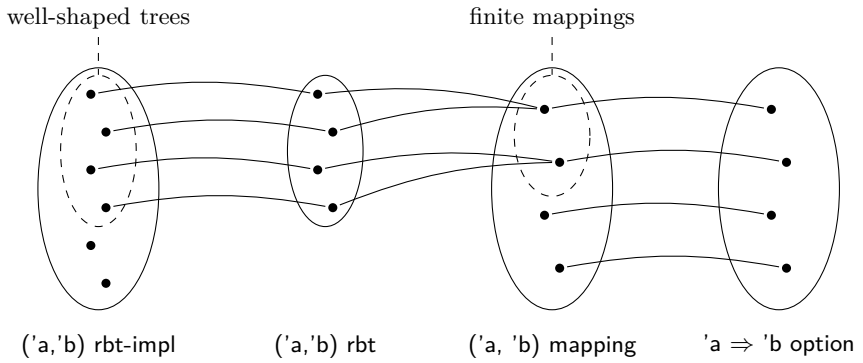


Fig. 3. Maps implemented by red-black trees

present in the type of h, but which still uses maps inside its body, you no longer need to lift h to some h', but you still have to prove a code equation for h itself that uses mappings internally. This is done again by transfer.

4.2 Using Lifting/Transfer

We can again use the Lifting and Transfer packages to automate the lifting. First, ('a, 'b) mapping is defined as a copy of 'a ⇒ 'b option and all primitive operations on maps are lifted:

```
typedef ('a, 'b) mapping = UNIV :: ('a ⇒ 'b option) set ..
setup_lifting(no_code) type_definition_mapping

lift_definition empty :: ('a, 'b) mapping is (λ_. None) .
lift_definition lookup :: ('a, 'b) mapping ⇒ 'a ⇒ 'b option is λm k. m k .
lift_definition update :: 'a ⇒ 'b ⇒ ('a, 'b) mapping ⇒ ('a, 'b) mapping
  is λk v m. m(k ↦ v) .
```

We showed only 3 such operations here but in reality there are more of them. Notice that we do not have to prove anything in the **lift_definition** command because the formal invariant preservation theorem is proved automatically if we work with type copies.

Now let us assume we used maps in our formalization to implement a special data type that behaves like a multiset and the multiplicity of elements is limited. Now we can implement an insert for this data structure that ensures that if the limit is reached, the map is not changed.

```
definition insert_lim :: ('a ⇒ nat option) ⇒ 'a ⇒ nat ⇒ 'a ⇒ nat option
where insert_lim m k lim = (case m k of
  Some n ⇒  if n < lim then m(k ↦ n + 1) else m
  | None ⇒ m(k ↦ 1))
```

We use **lift_definition** to define a copy of insert_lim that operates on the code generation type ('a, nat) mapping.

```
lift_definition insert_lim′ :: ('a, nat) mapping ⇒ 'a ⇒ nat ⇒ ('a, nat) mapping
  is insert_lim .
```

Function insert_lim′ is defined in terms of the original function insert_lim with the help of the morphisms between maps and mappings. In contrast to the situation in §3, we cannot use this definition as a code equation because it goes in the wrong direction: it reduces a computation on mappings to maps. The desired code equation for insert_lim′ is proved by transfer from the definition of the original function.

```
lemma [code]: insert_lim′ m k lim = (case Mapping.lookup m k of
  Some n ⇒ if n < lim then Mapping.update k (n + 1) m else m
  | None ⇒ Mapping.update k 1 m)
by transfer (fact insert_lim_def)
```

It is inconvenient that one has to write down the lifted code equation even if the proof is trivial thanks to Lifting/Transfer. In principle we can use the Transfer package to transfer goals in the other direction, i.e., from the concrete level to the abstract level and thus we would not have to write down the lifted code equation at all. But there is the problem that if we go in this direction, it is not clear which parts of a term should really be transferred. The transfer method can eagerly transfer all terms from the 'a ⇒ 'b option to the ('a, 'b) mapping level according to the transfer rules. But maybe the user would want some subterms to remain maps. This would require some mechanism that allows users to annotate a term and say which parts should not be transferred. This is work in progress and we intend to profit from the heuristics developed by Lammich [12]. Transferring from ('a, 'b) mapping to 'a ⇒ 'b option instead is unambiguous: all occurrences of mapping are replaced.

5 Applications

The following examples are the most important applications of data refinement in the Isabelle distribution.

Sets are implemented by lists by default. There is also an efficient implementation by red-black trees (in `Library/RBT_Set.thy`). In a recent application [19] a decision algorithm for MSO formulas was unusable with the default implementation of sets, but when theory `RBT_Set.thy` was loaded (no change of the client code is necessary!), it allowed us to decide small MSO formulas.

Mappings were described in §4. The distribution provides two implementations: red-black trees (as in §4) and association lists ('a × 'b) list.

Integers (type int) are defined as a total quotient of pairs of natural numbers nat × nat by the Lifting and Transfer packages. Two pairs of natural numbers (x, y) and (u, v) represent the same integer if $x + v = u + y$. We do not use this

definition for execution of integers because of efficiency reasons but we execute them by binary numerals num defined as a datatype:

datatype num = One | Bit0 num | Bit1 num

Operations on num are just usual binary arithmetic. Then, all integers are interpreted as binary numerals by employing three pseudo-constructors 0 :: int, Pos :: num ⇒ int and Neg :: num ⇒ int. The last step is to implement common integer operations by pattern-matching on these three pseudo-constructors and using corresponding operations on num.

Rationals (type rat) are defined as a partial quotient of pairs of integers int × int, again with the help of Lifting and Transfer. The quotient is partial because we do not include pairs (_,0) with a zero denominator. This is a logical definition of rational numbers used for formalizations but because the quotient is partial, we cannot use it directly for execution. Instead we interpret type rat as a subtype of int × int based on an observation that each rational number can be represented by a pair of co-prime integers with a non-zero denominator. Given the pseudo-constructor Frct :: int × int ⇒ rat, the *rep* function quotient_of :: rat ⇒ int × int is defined as follows

quotient_of r = (THE (n, d). r = Frct (n, d) ∧ $d > 0$ ∧ coprime n d)

and allows us to use the invariant mechanism described in §3 and execute rational numbers.

Reals (type real) are executed by rationals using the pseudo-constructor Ratreal :: rat ⇒ real and code equations for +, −, ∗, /, but not much more because only the rational reals are representable in this manner. But it is still useful. For example, it enables Quickcheck to find rational counterexamples to conjectures involving polynomials.

Basic arithmetic on complex numbers is executable without data refinement.

Outside the Isabelle distribution, data refinement has found a number of applications, too. For example, five entries in the *Archive of Formal Proofs* `http://afp.sf.org` define their own data abstractions, some of which are also discussed in the literature [14].

6 Related Work

Data refinement is a perennial topic that was first considered by Hoare more than 40 years ago [7], who already introduced abstraction functions and invariants. This principle of data refinement became an integral part of the model oriented specification language VDM [9] (and was later generalized to nondeterministic operations [15,6]). In the first-order context of universal algebra it was shown that there are always fully abstract models such that any concrete implementation can be shown correct with a homomorphism [16].

The infrastructure presented in this paper has been available in Isabelle for a few years but has never been published properly: [5] merely shows an example (similar to §2); the core of the present paper, the treatment of invariants as in

§3, was not available at that time. Nevertheless the infrastructure has already been used in many places (see §5). Based on this infrastructure, Lochbihler [13] has recently overcome some limitations of our approach (e.g., see the end of §2).

Lammich [12] has implemented a new framework for data refinement that has some similarity with §4: you do not obtain code for f but for some f' that is in a certain relationship to f. As a result he can work with a more general notion of refinement supporting (for example) nondeterministic operations and multiple implementations of the same type. Of course he also faces the complications explained in §4.1. A difference is that his system proves invariance preservation for derived functions as an explicit theorem whereas for us the type checker does the work. In a nutshell, his is a general framework for heavy duty data refinement, ours is a lightweight infrastructure for completely transparent but more limited data refinement.

ACL2 supports data refinement, too. In the Mu-calculus case study [11] Manolios shows how to implement sets by lists (using a congruence on lists) whereas Greve *et al.* [4] explain how to deal with invariants. The details are rather different from our work because ACL2 is untyped. In Coq [1], parametrized modules support a form of data refinement [3]: perform your development inside the context of a specification of finite sets (or whatever abstract type you have), and later instantiate the module with some implementation of finite sets that has been proved to satisfy the finite set axioms. The drawback is that you do not really work with the actual abstract type (e.g., sets), but some axiomatization of it, which may not have the same nice syntax and proof support.

7 Conclusion

We have presented Isabelle/HOL's infrastructure for a lightweight approach to data refinement. Its distinctive feature is the tight integration with the code generator and hence also any tool that builds on it. The key principle is that when you want to execute function f, you really execute f, which in turn calls a more efficient implementation f' that was proved equivalent to f. As a result, data refinement is completely transparent to the user: just load a specific refinement theory and the code generator does the rest. This completely automatic approach assumes that refinement happens on a per type constructor basis. To remove this assumption we presented a more general approach that sacrifices some of the above advantages. It relies strongly on two packages for lifting definitions and theorems from one type to another automatically.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
2. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011)

3. Filliâtre, J.-C., Letouzey, P.: Functors for Proofs and Programs. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 370–384. Springer, Heidelberg (2004)
4. Greve, D., Kaufmann, M., Manolios, P., Moore, J., Ray, S., Ruiz-Reina, J., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Functional Programming 18, 15–46 (2008)
5. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
6. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
7. Hoare, C.A.R.: Proof of Correctness of Data Representations. Acta Informatica 1, 271–281 (1972)
8. Huffman, B., Kunčar, O.: Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. Presented at the Isabelle Users Workshop at ITP 2012 (2012), http://www21.in.tum.de/~kuncar/huffman-kuncar-itp2012.pdf
9. Jones, C.B.: Software Development. A Rigourous Approach. Prentice Hall (1980)
10. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C.-C. (eds.) Proc. of the 26th ACM Symposium on Applied Computing (SAC 2011), pp. 1639–1644. ACM (2011)
11. Kaufmann, M., Manolios, P., More, J.S.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
12. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Heidelberg (2013)
13. Lochbihler, A.: Light-weight containers for Isabelle: efficient, extensible and nestable. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 116–132. Springer, Heidelberg (2013)
14. Lochbihler, A., Bulwahn, L.: Animating the Formalised Semantics of a Java-like Language. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 216–232. Springer, Heidelberg (2011)
15. Nipkow, T.: Non-Deterministic Data Types: Models and Implementations. Acta Informatica 22, 629–661 (1986)
16. Nipkow, T.: Are Homomorphisms Sufficient for Behavioural Implementations of Deterministic and Nondeterministic Data Types? In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 260–271. Springer, Heidelberg (1987)
17. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Reif, W., Schellhorn, G., Stenzel, K.: Interactive Correctness Proofs for Software Modules Using KIV. In: COMPASS 1995: Proc. Tenth Annual Conf. Computer Assurance, pp. 151–162. IEEE (1995)
19. Traytel, D., Nipkow, T.: A Verified Decision Procedure for MSO on Words (2013), http://www.in.tum.de/~nipkow/pubs

Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable

Andreas Lochbihler

Institute of Information Security, ETH Zurich
andreas.lochbihler@inf.ethz.ch

Abstract. In Isabelle/HOL, we develop an approach to efficiently implement container types such as sets and maps in generated code. Thanks to type classes and refinement during code generation, our light-weight framework is flexible, extensible, and easy to use. To support arbitrary nesting of containers, we devise an efficient linear order on sets that can even compare complements and non-complements. Our evaluation shows that it is both efficient and usable.

1 Introduction

Recently, executable implementations have been generated from increasingly large developments in theorem provers. Early works [3,8,16] implemented containers inefficiently, in particular as lists and closures, or burdened the formalisation with complex data structure details. Today, refinement approaches [5,6,9,10] enable the verification to operate on abstract types like sets and functions – for code generation, the refinement replaces them with efficient implementations. For use in large-scale projects, they should meet four requirements:

ease of use. It requires little effort to apply the refinement to an application.

flexibility. Applications themselves can choose which implementations to use for which container, and can easily switch between them; multiple implementations for the same container type are supported simultaneously.

extensibility. When a user adds another implementation or a new type of stored data, he need not touch the existing parts. This is crucial for modularisation: Libraries can be included unchanged and extended incrementally.

nesting. Containers can be nested arbitrarily, e.g., a map from sets to sets of sets.

For the proof assistant Isabelle/HOL and its code generator [7], the existing approaches differ in where the refinement happens. On the one hand, the Isabelle Collections Framework (ICF) [9] explicitly models refinement inside the logic: it defines a uniform interface to various verified data structures. It meets the above criteria except for ease of use and nesting. First, users must manually introduce copies for all definitions such that they use the interface and prove refinement subject to invariants of the data structure. In practice, refinement can make up a substantial part of the development [15]. Second, Isabelle's proof automation assumes unique representation of objects, but ICF-style refinement introduces multiple ones, e.g., both lists $[a, b]$ and $[b, a]$ implement the same set

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 116–132, 2013.

$\{a, b\}$. So, at present, the ICF does not support a set of sets implemented as unordered lists or red-black trees (RBT) – although demand for nested sets has been expressed on the Isabelle mailing list, e.g., posts 4D779F63.9050506@irit.fr and 89A2F83C-2AE8-44B7-B1ED-5EDD4E160C1F@loria.fr.

On the other hand, Haftmann et al. [6] advocate for refinement inside the code generator, see §1.1 for an introduction. This is easy to use, as the user need not bother about the details of implementations. In particular, representations remain unique in the logic, so nesting is possible. Yet, the current state in Isabelle2013 is far from fully exploiting the available features. For example, it supports only one implementation for each type, i.e., *all* sets are implemented either as lists or as RBTs. In a case study on extracting a Java interpreter [15], the elements of *some* sets could not be ordered linearly (as required for RBTs), so *all* sets were inefficiently implemented as lists.

Contributions. We present an easy to use approach (called LC for light-weight containers) based on Haftmann's [6]. It supports multiple implementations and is flexibile, extensible, and nestable (§2). We devise a configurable scheme to automatically choose a suitable implementation based on the type of what is to be stored (§2.3). To avoid type class restrictions, we adapt a Haskell approach with type constructor classes [17] to Isabelle's single-parameter type classes. For the presentation, we focus on sets implemented as closures, lists, distinct lists, or RBTs. As our development covers all set operations from Isabelle, it can replace the default setup for code generation in applications without much ado. We have also covered maps, i.e., support the two fundamental container types.

Second, to support arbitrary nesting of sets, we devise an efficient linear order on sets (§3). It requires only linearly many comparisions between the elements and supports comparisons even between complements and non-complements. As Isabelle's automated disprover `quickcheck` [2] relies on code generation, it is important to support complements, too.

Third, we evaluate our approach in two micro-benchmarks and a case study (§4). The benchmarks show that our approach generates code as efficient as the ICF and that the linear order on sets is also efficient. The case study with the Java interpreter shows that our approach integrates seamlessly with the existing Isabelle setup and is therefore as easy to use; switching from Isabelle's default setup to ours did not require any adaptations. Moreover, multiple implementations for sets and maps improve the execution times.

Moreover, our way of combining the powerful features of the code generator is novel; we describe the ideas in §5. We would like to stress that we have proven all lemmas and refinements in this paper in Isabelle; LC is available online [14].

1.1 Background: The Code Generator Framework and Refinement

Isabelle's code generator [6,7] turns a set of equational theorems into a functional program with the same equational rewrite system. The translation guarantees partial correctness by construction, as it builds on equational logic.

Program refinement separates code generation issues from the rest of the formalisation in Isabelle. As any (executable) equational theorem suffices for

code generation, the user may *locally* derive new (code) equations to use upon code generation. Hence, existing definitions and proofs remain unaffected.

For *data refinement*, the user replaces constructors of a datatype by other constants and derives equations that pattern-match on these new (pseudo-)constructors. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type. Again, this is local as it affects only code generation, but not the logical properties of the refined type. Thus, one cannot exploit the type's new structure inside the logic.

For example, Isabelle's default code generator setup represents sets (type α *set*) with the two pseudo-constructors *set* and *coset* of type α *list* $\Rightarrow \alpha$ *set*, which represent the (complement of the) finite set of elements in the given list. Note that they are neither injective (*set* $[1,2] =$ *set* $[2,1]$, but $[1,2] \neq [2,1]$), nor do they exhaust the type α *set* if α is infinite (e.g., $\{1,3,5,\ldots\}$), nor are they disjoint if α is finite (*set* [*True*] = *coset* [*False*] for booleans). Nevertheless, the following equations implement the membership test $\in$ on type α *set*:

$$(x \in \mathit{set}\ xs) = (\mathit{memb}\ xs\ x) \qquad (x \in \mathit{coset}\ xs) = (\neg\, \mathit{memb}\ xs\ x) \qquad (1)$$

where *memb* xs x checks if x equals one of xs's elements by traversing xs – the type class *equal* provides the implementation of these equality tests.

This is an example of *sort refinement*: The equality test in *memb*'s code equations requires that α is of sort *equal*, but *memb*'s specification in the logic as $\lambda xs\ x.\ x \in \mathit{set}\ xs$ does not. The code generator collects, propagates, and checks all sort constraints upon code generation. Thus, it propagates $\alpha :: \mathit{equal}$ via (1) to $\in$, too. Sorts intersect: Suppose that another pseudo-constructor *tree* represents finite sets as binary search trees and we use $(x \in \mathit{tree}\ t) = (\mathit{lookup}\ t\ x \neq \mathit{None})$, where *lookup* obtains the linear order on α from the type class *linorder*. Then, the code generator enforces that any invocation of $\in$ operates on sets whose element type instantiates both *equal* and *linorder*.

1.2 Related Work

The ICF approach [9] considers refinement inside the logic superior to refinement in the code generator, as the implementation can exploit the refined structure and, e.g., resolve non-determinism from underspecification such as the order of iteration over a set. However, the ICF requires more adaptations; some automation has been developed for monadic programs [10], but this still requires adapting the application to the refinement calculus. We argue that both approaches complement each other, and recommend to use the ICF only when necessary, and to stick with the simpler refinement in the code generator whenever possible. For example, in [15], we used Haftmann's approach when sets and maps are accessed only through membership tests and lookup operations, respectively, and the ICF to resolve non-determinism, e.g., to pick an arbitrary element from a set.

Peyton Jones [17] and Chen et al. [4] show that Haskell's single-parameter type classes do not suffice for bulk-type polymorphism (flexibility and extensibility in our terminology). Our approach nevertheless succeeds, because refinement is incremental and we do not have to extend the generated code itself.

Lescuyer's Containers library [11] for Coq efficiently implements finite sets and maps with type classes, but he cannot represent set complements. His linear order on sets is based on the lexicographic ordering of the sorted list of elements.

2 Multiple Implementations for Containers

Sort intersection as described in §1.1 creates problems in large-scale applications that use the same HOL container type such as α *set* for different α, as some element types α fail to meet all sort constraints. In previous work [15], e.g., some sets contain strings (a sequence of characters) and others functions. Unfortunately, we could not make strings an instance of *linorder*, because the order on lists had already been fixed to the partial prefix order elsewhere, and functions lack computable equality for the type class *equal*. Thus, we had to stick with inefficient lists which allow duplicates, see §4.3 for details.

In this section, we introduce new type classes which *any* type can be made an instance of and show how to support multiple implementations (§2.1). Thus, sort intersection is no longer a show-stopper. Then, we demonstrate extensibility by adding new data and a new implementation (§2.2). To improve usability, we devise a configurable scheme for automatically choosing an implementation (§2.3) and show how to deal with binary operations (§2.4).

2.1 New Type Classes and Multiple Implementations

To avoid instantiation obstacles, Peyton Jones [17, §3] introduces new type classes whose parameters already tell whether the operation is supported. Here, we borrow his idea and adapt it to the theorem prover setting. We introduce a new type class *ceq* for equality on container elements (to make the overloading explicit, we write the type parameter as a superscript to type class parameters):

$$\begin{aligned}\textbf{class}\ ceq = \text{fixes}\ & ceq^{\alpha} :: (\alpha \Rightarrow \alpha \Rightarrow bool)\ option \\ \text{assumes}\ & ceq^{\alpha} = \lfloor eq \rfloor \Longrightarrow eq = (op =)\end{aligned}$$

The declared equality operator ceq^{α} tells whether elements of type α may be tested for equality at all. If so ($ceq^{\alpha} = \lfloor eq \rfloor$ for some eq; $\lfloor \_ \rfloor$ denotes definedness), the assumption enforces that eq in fact implements HOL equality $op =$ (we forbid other congruence relations, as Isabelle's proof automation cannot handle them well). Otherwise ($ceq^{\alpha} = None$), *ceq* does not impose any constraints on the implementation and – as all proofs rely only on the specified assumptions – neither must any usage of ceq^{α}. Thus, *every* type can be made an instance of *ceq*. For example, the instantiations for the function space constructor *fun* (written infix as $\Rightarrow$) and natural numbers *nat* are as follows:

instantiation *fun* :: (*type*, *type*) *ceq* begin
definition $ceq^{\alpha \Rightarrow \beta} = None$
instance ⟪proof⟫ end

instantiation *nat* :: *ceq* begin
definition $ceq^{nat} = \lfloor op = \rfloor$
instance ⟪proof⟫ end

Unfortunately, we cannot follow Peyton Jones' development any further: he introduces a type constructor class for collection type constructors that specifies

the operations, but Isabelle does not support type constructor classes. Instead, we use data refinement. For the moment, we only consider three implementations: lists with and without duplicates and characteristic functions; in §2.2, we add efficient RBTs. To that end, we define four pseudo-constructors for α *set* that replace *set* and *coset* for the code generator:

char. function	$ChF :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$	$ChF\ P = \{ x.\ P\ x \}$
monad style	$MSet :: \alpha\ list \Rightarrow \alpha\ set$	$MSet\ xs = set\ xs$
distinct list	$DSet :: \alpha\ dlist \Rightarrow \alpha\ set$	$DSet\ ds = \{ x.\ dmemb\ ds\ x \}$
complement	$Compl :: \alpha\ set \Rightarrow \alpha\ set$	$Compl\ A = \overline{A}$

For α of sort *ceq*, the type α *dlist* consists of all lists from α *list* whose elements are pairwise distinct w.r.t. the equality operator from *ceq*; if ceq^α is undefined, α *dlist* consists of all lists. *dmemb ds x* checks if x occurs in *ds* using *ceq*'s equality operator. We model the complement of a set A (notation $\overline{A}$) as *Compl A*.

Monad-style sets (*MSet*) can be used to model non-determinism. They allow duplicates and avoid equality checks whenever possible, e.g., for *insert*, $\cup$, and *bind*. Still, we do implement operations that require equality, but they may fail at run time when ceq^α is *None*. Membership, e.g., uses the following equation:

$$\begin{aligned}(x \in MSet\ xs) = (case\ ceq^\alpha\ of\ None \Rightarrow error\ (\lambda\_.\ x \in MSet\ xs)\\ \mid \lfloor eq \rfloor \Rightarrow memb'\ eq\ xs\ x)\end{aligned} \tag{2}$$

where *error* logically returns its argument applied to the unit value (), but it raises an exception in the generated code at run time; the unit closure ensures termination in call-by-value languages like ML. As *memb'* takes the equality operation as a parameter, we do not depend on the type class *equal*. Note that membership $\in$ cannot be total, as we deliberately do not require an implementation for equality. Like in the common case of missing patterns in functional programs, the user himself must ensure that ceq^α is defined for the element type α whenever he calls $\in$ on *MSet*. The other pseudo-constructors do not need such a run time check, as we have defined them logically in terms of membership:

$$(x \in ChF\ P) = P\ x \qquad (x \in DSet\ ds) = dmemb\ ds\ x \qquad (x \in Compl\ A) = (x \notin A)$$

2.2 Extensibility

Extensibility expresses that one may use containers with new types of elements and add new implementations for a container without editing the existing code base. This ensures that users can freely extend a container framework. In this section, we demonstrate that our light-weight approach achieves this.

To use a new type of elements, one merely has to instantiate the new type classes, i.e., *ceq* in the example. As the operations can default to *None*, this is always possible. For example, arithmetic expressions:

datatype *expr* = *Val nat* | *Var string* | *Plus expr expr* | *Times expr expr*
instantiation *expr* :: *ceq* begin
definition $ceq^{expr} = \lfloor op = \rfloor$ Note that the datatype declaration already generates code equations for *op* =.
instance ⟨⟨proof⟩⟩ end

Next, we show how to add an implementation of sets backed by RBTs. This requires a linear order on container elements, i.e., a new type class *corder* (where *class.linorder leq lt* denotes that *leq* is a linear order and *lt* its strict version):

$$\begin{aligned}&\textbf{class } \mathit{corder} = \text{fixes } \mathit{corder}^{\alpha} :: ((\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}) \times (\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}))\ \mathit{option}\\ &\qquad\qquad \text{assumes } \mathit{corder}^{\alpha} = \lfloor (\mathit{leq}, \mathit{lt}) \rfloor \Longrightarrow \mathit{class.linorder}\ \mathit{leq}\ \mathit{lt}\end{aligned}$$

Our RBT implementation builds on the verified RBT formalisation in Isabelle's library, which is based on the *linorder* type class; we only have to adapt it to *corder*. Analogous to α *dlist*, the type α *srbt* of RBT sets contains all RBTs that are sorted according to *corder*, if *corder* is defined; otherwise, it contains all binary trees irrespective of sorting and balancing. Then, we define the new pseudo-constructor $\mathit{RSet}\ rs = \{\, x.\ \mathit{rmemb}\ rs\ x \,\}$ where *rmemb* denotes the lookup operation on α *srbt*, which uses *corder* for comparing elements. To finish, we declare *RSet* as another pseudo-constructor for α *set* and prove code equations for all set operations. Again, operations like *insert* are correctly implemented only if *corder* implements some linear order; otherwise, they fail with an exception during execution.

$$\begin{aligned}(x \in \mathit{RSet}\ rs) &= \mathit{rmemb}\ rs\ x\\ \mathit{insert}\ x\ (\mathit{RSet}\ rs) &= (\mathit{case}\ \mathit{corder}^{\alpha}\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{error}\ (\lambda\_.\ \mathit{insert}\ x\ (\mathit{RSet}\ rs))\\ &\qquad \mid \lfloor \_ \rfloor \Rightarrow \mathit{RSet}\ (\mathit{rinsert}\ x\ rs))\end{aligned} \quad (3)$$

Sort refinement requires that from now on all element types inhabit *corder*, too, as α has sort *corder* in (3). Hence, we instantiate *corder* for the element types. Since it is very similar to the *linorder* type class, the instantiations are canonical and full Isabelle support is available, e.g., Thiemann's order generator for data types [19]. For example, the proofs in the following are automatic.

```
instantiation nat :: corder begin
definition ceq^nat = ⌊(op ≤, op <)⌋
instance ⟨proof⟩ end
```

```
derive linorder expr
instantiation expr :: corder begin
definition ceq^expr = ⌊(op ≤, op <)⌋
instance ⟨proof⟩ end
```

2.3 Automatically Choosing an Implementation

Recall from (2) and (3) that the generated code raises a run-time exception in case of an unsupported operation. One can analyse the code equations to that end before code generation, but we have not yet implemented such an analysis. Instead, we let the generated code choose the implementation based on the operations the element type provides, e.g., use *RSet* only if corder^{α} is defined. Then, we are sure that the necessary operations are available, i.e., the exception in (3) cannot occur. However, exceptions can still occur when a set operation has no implementation at all. For a set of functions, e.g., $\in$ will still fail in (2). This is a design choice: We want to support α *set* for *all* α. If α does not permit to implement a set operation at all, it is the user's fault to apply the operation to α *set*.

Peyton Jones [17, §3.3] proposes a similar approach, but his is neither extensible nor flexible. He avoids the implementability problem by requiring equality for all element types, but this does not fit to how sets are used in Isabelle.

All sets originate from either a set comprehension *ChF P* or the empty set $\emptyset$. We ignore the pseudo-constructor *ChF*, as the operations for *ChF* do not depend on *ceq* or *corder*. For $\emptyset$, our code equations heuristically pick an implementation. The following is a first, naïve attempt in the style of [17]:

$$\begin{aligned}\emptyset = (&\mathit{case}\ \mathit{corder}^{\alpha}\ \mathit{of}\ \lfloor\_\rfloor \Rightarrow \mathit{RSet}\ \mathit{rempty}\\ &\mid \mathit{None} \Rightarrow \mathit{case}\ \mathit{ceq}^{\alpha}\ \mathit{of}\ \lfloor\_\rfloor \Rightarrow \mathit{DSet}\ \mathit{dempty} \mid \mathit{None} \Rightarrow \mathit{MSet}\ []) \end{aligned} \tag{4}$$

This equation uses RBTs if there is a linear order on the elements. Otherwise, it tries distinct lists for equality and picks monad-style sets as last resort. This way, RBTs are only used for element types that do provide a linear order *corder*. Thus, the error in (3) cannot trigger. Nevertheless, we cannot eliminate the check, as the user still can misuse *RSet* in his own equations. However, (4) offers too little control over the choice and thus violates flexibility. For some types, it is sensible to use distinct lists even if there is a linear order – for *bool* with just two elements, e.g., the *DSet* implementation is four times faster than the *RSet* one.[1]

Instead, we let an overloaded operation choose the implementation:

$$\textbf{class}\ \mathit{set\text{-}impl} = \text{fixes}\ \mathit{set\text{-}impl}^{\alpha} :: \mathit{set\text{-}impl}$$

The type *set-impl* (inhabited by only one value *Set-IMPL*) has a pseudo-constructor for each implementation: *Set-ChF*, *Set-dlist*, *Set-RBT*, *Set-Monad*, plus *Set-Auto* for automatic selection like in (4). They are only pseudo-constructors such that we can add more for new implementations later. Then, we implement $\emptyset$ via $\emptyset$ = *sempty set-impl*$^{\alpha}$, where the function *sempty*, logically defined by *sempty Set-IMPL* = $\emptyset$, chooses the desired implementation:

$$\begin{aligned}&\mathit{sempty}\ \mathit{Set\text{-}ChF} = \mathit{ChF}\ (\lambda\_.\ \mathit{False}) && \mathit{sempty}\ \mathit{Set\text{-}dlist} = \mathit{DSet}\ \mathit{dempty}\\ &\mathit{sempty}\ \mathit{Set\text{-}RBT} = \mathit{RSet}\ \mathit{rempty} && \mathit{sempty}\ \mathit{Set\text{-}Monad} = \mathit{MSet}\ []\\ &\mathit{sempty}\ \mathit{Set\text{-}Auto} = (\mathit{case}\ \mathit{corder}^{\alpha}\ \mathit{of}\ \lfloor\_\rfloor \Rightarrow \mathit{RSet}\ \mathit{rempty}\ \mid\ \ldots)\end{aligned}$$

Note that we could have overloaded $\emptyset$ directly without the detour *set-impl* and *sempty*. Yet, as Isabelle allows overloading only for constants with exactly one type parameter, this does not extend to other container types like maps with multiple type parameters. Our approach also works such container types.

Below, we give three example instantiations for *set-impl*. As motivated above, *bool* chooses distinct lists although *corder*^{bool} is defined. α *option* (the type of $\lfloor\_\rfloor$ and *None*) inherits the choice from its type argument, as it adds only one value. In contrast, a set of sets discards any preference from the element type and falls back on automatic selection. This seems sensible, as a set of sets can become much larger than a set of the elements.

$$\mathit{set\text{-}impl}^{bool} = \mathit{Set\text{-}dlist} \qquad \mathit{set\text{-}impl}^{\alpha\ option} = \mathit{set\text{-}impl}^{\alpha} \qquad \mathit{set\text{-}impl}^{\alpha\ set} = \mathit{Set\text{-}Auto}$$

[1] Build, e.g., the set $\{\mathit{True}, \mathit{False}\}$ and check membership for both elements. Under PolyML, 1M (10M) iterations take .05 s (.47 s) for *DSet* and .21 s (2.05 s) for *RSet*.

Moreover, users can later change the implementations for a type α, if they want to: as all pseudo-constructors are logically equivalent, proving a different code equation for $\textit{set-impl}^{\alpha}$ is straightforward: As all pseudo-constructors are logically equal to *Set-IMPL*, we can, e.g., prove $\textit{set-impl}^{\alpha\ \textit{option}} = \textit{Set-ChF}$ and use this code equation to choose characteristic functions as default for α *option*.

2.4 Binary Operations

Binary operations like $\cap$ and $\cup$ require pattern-matching on both sets, i.e., the number of possible combinations grows quadratically with the number of implementations. In the ICF [9], a Ruby script automatically generates implementations for all combinations, all of which use a generic implementation parametrised by iterators and basic operations. More efficient implementations for special combinations (like intersecting two *RSet*s [1]) are not supported.

With our approach, sequential pattern matching in the target language offers a better solution. First, we derive general equations that pattern-match only on one constructor and compute the result generically. Second, we show equations for special cases with more efficient implementations, which take precedence over the generic ones as pattern matching is sequential. This keeps the number of equations linear in the number of implementations plus the optimised cases. Moreover, the general equations automatically cover future set implementations.

For intersection and *RSet*, e.g., we obtain the following, where *rfilter* $P\ rs$ retains only rs's elements that satisfy the predicate P and *rint* is the fast intersection algorithm on α *srbt*.

$$\textit{RSet}\ rs_1 \cap \textit{RSet}\ rs_2 = (\textit{case}\ \textit{corder}^{\alpha}\ \textit{of None} \Rightarrow \ldots \mid \lfloor\_\rfloor \Rightarrow \textit{RSet}\ (\textit{rint}\ rs_1\ rs_2))$$
$$\textit{RSet}\ rs \cap A = (\textit{case}\ \textit{corder}^{\alpha}\ \textit{of None} \Rightarrow \ldots \mid \lfloor\_\rfloor \Rightarrow \textit{RSet}\ (\textit{rfilter}\ (\lambda x.\ x \in A)\ rs))$$
$$A \cap \textit{RSet}\ rs = (\textit{case}\ \textit{corder}^{\alpha}\ \textit{of None} \Rightarrow \ldots \mid \lfloor\_\rfloor \Rightarrow \textit{RSet}\ (\textit{rfilter}\ (\lambda x.\ x \in A)\ rs))$$

When we prove these equations, we cannot exploit sequentiality of pattern matching, i.e., we implicitly prove that all right-hand sides are equal when the left-hand sides unify. This is only possible as refinement happens in the code generator, i.e., our pseudo-constructors abstract from the concrete representation. As the ICF models the refinement in the logic, it cannot prove such equalities.

3 Executable Linear Order on Sets

Recall that our approach abstracts from different implementations in the logic. Thus, it directly supports arbitrary nesting of containers, provided that we make the container type an instance of the type classes – in our example, *ceq* and *corder* for α *set*. Not to lose on efficiency, we now devise a linear order $\sqsubseteq$ on sets and implement $\textit{corder}^{\alpha\ \textit{set}} = \lfloor(\sqsubseteq, \sqsubset)\rfloor$. This is one example where the separate type class *corder* is crucial: As Isabelle fixes the canonical order $\leq$ on α *set* to the non-linear subset order $\subseteq$, we cannot make α *set* an instance of *linorder*.

By the axiom of choice, there is a linear order on every set, but we cannot implement this order, so it is useless here. Fortunately, it suffices if we can decide

the ordering on representable sets. Given a linear order $\leq$ on the elements, we first define a linear order on the finite and cofinite sets. Then, we extend it to a linear order on all sets by the axiom of choice (§3.1), as *corder* requires a linear order on all elements. Our equations for code generation (§3.2) pattern-match on the pseudo-constructors which represent only finite or cofinite sets. If the (co)finite sets are given as sorted lists of their (non-)elements, $\sqsubseteq$ requires at most linearly (in the size of the lists) many $\leq$-comparisons – except for comparing a finite and a cofinite set when α is finite, because a set may be both finite and cofinite. For the latter case, we show that further operations on α are necessary, and implement them for α *set* to ensure nestability and extensibility (§3.3).

Moreover, our order $\sqsubseteq$ satisfies the following properties for all sets A, A' and finite sets F, F', which facilitate proving the code equations in §3.2:

(P1) $\emptyset \sqsubseteq A$ and $A \sqsubseteq \mathit{UNIV}$
(P2) If $F \subseteq F'$, then $F \sqsubseteq F'$
(P3) $\overline{A} \sqsubseteq \overline{A'}$ iff $A' \sqsubseteq A$
(P4) If α is infinite, then $F \sqsubseteq \overline{F'}$

Properties P1 and P2 describe the similarity with the subset order: the empty set $\emptyset$ and the full set *UNIV* of all elements are the least and greatest sets, resp., and $\sqsubseteq$ extends $\subseteq$ on finite sets. Hence, when iterating over a set of finite sets in ascending order, one visits subsets before supersets. P3 allows to drop the *Compl* constructor on both sides if the relation is reversed, i.e., complement is anti-monotone. P4 expresses that finite sets are always less than cofinite sets, if α's universe is infinite. If α is finite, $F \sqsubseteq \overline{F'}$ is inconsistent with P2 ($\emptyset \subseteq \{ a \} \sqsubseteq \overline{\mathit{UNIV}} = \emptyset$, i.e., $\{ a \} = \emptyset$, a contradiction), because then sets are both finite and cofinite.

3.1 Definition

We construct our linear order $\sqsubseteq$ from two intermediate partial orders $\sqsubseteq_1$ and $\sqsubseteq_2$, where each step extends the previous order.

First, we define that $A \sqsubset_1 B$ holds whenever both A and B are finite and B contains the mininum element of the symmetric difference of A and B. Intuitively, if $A = \mathit{set}\ as$ and $B = \mathit{set}\ bs$ with as and bs duplicate-free and $\leq$-sorted in ascending order, $A \sqsubset_1 B$ iff as precedes bs in the lexicographic list order w.r.t. the converse $\geq$ of $\leq$ (Lem. 1). For a three-value type with order $0 < 1 < 2$, e.g., $\sqsubset_1$ orders the sets as follows:

$$\emptyset \sqsubset_1 \{ 2 \} \sqsubset_1 \{ 1 \} \sqsubset_1 \{ 1, 2 \} \sqsubset_1 \{ 0 \} \sqsubset_1 \{ 0, 2 \} \sqsubset_1 \{ 0, 1 \} \sqsubset_1 \{ 0, 1, 2 \} \quad (5)$$

Taking the converse of $\leq$ is crucial for the above properties. In the example, the lexicographic list order w.r.t. $\leq$ would give $\{ 0, 1, 2 \} \sqsubset_1 \{ 1 \}$, which violates P1 and P2. Moreover, note the symmetry with complements in (5): the n-th set from the left is the complement of the n-th set from the right.

For finite types like in (5), $\sqsubset_1$ completely determines $\sqsubseteq$. So let us move on to infinite types. Fix a set of sets $\mathfrak{C} :: \alpha\ \mathit{set}\ \mathit{set}$ with two properties: (i) If $A :: \alpha\ \mathit{set}$ is finite, then $A \in \mathfrak{C}$. (ii) If α is infinite, then $A \in \mathfrak{C}$ iff $\overline{A} \notin \mathfrak{C}$. Such a $\mathfrak{C}$ exists: If α is finite, take *UNIV*. Otherwise, consider the subset order restricted to sets

of sets that satisfy (i) and the "only if" direction of (ii). For any chain in this order, the union of the chain's sets of sets is an upper bound. Thus, by Zorn's lemma, the order has a maximal element – and all maximal elements satisfy (i) and (ii). For infinite α, the set $\mathfrak{C}$ decides for each set A if $\sqsubseteq$ treats it like a finite set ($A \in \mathfrak{C}$) or like a cofinite set ($A \notin \mathfrak{C}$). If A is infinite, this decision does not matter, as we do not care about infinite sets, but it ensures complement symmetry P3.

Next, $\sqsubset_2$ extends $\sqsubset_1$ to a linear order on $\mathfrak{C}$ such that $\emptyset$ is the least element. Hence, $\emptyset \sqsubseteq_2 A$ even if $A \in \mathfrak{C}$ is infinite. By the order extension principle, $\sqsubset_2$ exists. This suffices for our purpose, as we care only about finite sets, i.e., we need not specify $\sqsubset_2$ completely for infinite A and B.

Finally, we mirror $\sqsubset_2$ at the boundary of $\mathfrak{C}$ to obtain complement symmetry P3. We define $\sqsubset$ as follows:

$$A \sqsubset B = (\mathit{if}\ A \in \mathfrak{C}\ \mathit{then}\ A \sqsubset_2 B \vee B \notin \mathfrak{C}\ \mathit{else}\ B \notin \mathfrak{C} \wedge \overline{B} \sqsubset_2 \overline{A})$$

Property P4 holds as the cofinite sets, which are not in $\mathfrak{C}$, are greater than the finite ones (which are members of $\mathfrak{C}$). Note that if α is finite, $\mathfrak{C} = \mathit{UNIV}$ and therefore, $\sqsubset$ is identical to $\sqsubset_2$ and $\sqsubset_1$.

3.2 Code Equations

Now, we derive code equations to implement $\sqsubset$ and $\sqsubseteq$ for finite and cofinite sets. In the following, we assume that a (co)finite set is given as a sorted and duplicate-free list of (the complement's) elements. Thanks to the above properties, some combinations are straightforward: P3 reduces comparisons between two cofinite sets to comparing their complements; and P4 already decides comparisons between one finite and one cofinite set if α is infinite. Thus, only two cases are left: comparing two finite sets and – if α is finite – comparing a finite and a cofinite set.

For the first case, we show that $\sqsubset_1$ is a kind of lexicographic ordering:

Lemma 1. *Let A and B be non-empty, finite sets and let $\mathit{Min}\ A$ and $\mathit{Min}\ B$ denote their respective minimum elements. Then, $A \sqsubset_1 B$ iff $\mathit{Min}\ A > \mathit{Min}\ B$, or $\mathit{Min}\ A = \mathit{Min}\ B$ and $A - \{\mathit{Min}\ A\} \sqsubset_1 B - \{\mathit{Min}\ B\}$.*

Corollary 1. *Let xs and ys be sorted, duplicate-free lists. Then, set $xs \sqsubset_1$ set ys iff lexord $(>)$ xs ys, where lexord $(>)$ is the lexicographic order on lists for the element order $>$.*

Thus, we can implement comparisons between finite sets efficiently as a lexicographic order. If we store the sets in sorted order (like RBTs do), the number of element comparisons is linear in the size of the sets.

Now, only comparisons between a finite and a cofinite set remain if α is finite. Unfortunately, we cannot (computationally) decide such comparisons solely by looking at the representations of finite and cofinite sets. This is not a fault of our choice for $\sqsubset$, but impossible for any linear order $\Subset$ implemented as a polymorphic function of type $\alpha\ \mathit{set} \Rightarrow \alpha\ \mathit{set} \Rightarrow \mathit{bool}$. To see this, compare the sets $\overline{\{a\}}$ and $\emptyset$. If

a is the only value that inhabits α, $\overline{\{\,a\,\}} = \emptyset$ and therefore $\overline{\{\,a\,\}} \notin \emptyset$ and $\emptyset \notin \overline{\{\,a\,\}}$, as $\Subset$ is irreflexive. Otherwise, $\overline{\{\,a\,\}} \neq \emptyset$ and thus $\overline{\{\,a\,\}} \Subset \emptyset$ or $\emptyset \Subset \overline{\{\,a\,\}}$ by linearity. Thus, $\Subset$ must know whether α contains further values, but it cannot compute that solely from its arguments $\{\,a\,\}$ and $\overline{\emptyset}$. Similar examples show that $\Subset$ has to know if there are further values above, below, or between any two values.

Therefore, we introduce another overloaded operation *proper interval* pi^α of type $\alpha\ option \Rightarrow \alpha\ option \Rightarrow bool$ that checks whether an open interval is proper, i.e., non-empty. Intervals are given by their bounds, *None* represents unboundedness. Hence, all implementations of pi^α satisfy the following specification (note that $(\exists z.\ True) = True$ as all HOL types are inhabited):

$$\begin{array}{ll} pi^\alpha\ None\ None = True & pi^\alpha\ None\ \lfloor y \rfloor = (\exists z.\ z < y) \\ pi^\alpha\ \ \lfloor x \rfloor\ \ None = (\exists z.\ x < z) & pi^\alpha\ \ \lfloor x \rfloor\ \ \lfloor y \rfloor = (\exists z.\ x < z \wedge z < y) \end{array} \tag{6}$$

Now, we present the case of comparing a complement with a non-complement. To decide $\overline{A} \sqsubseteq_1 B$, where $A = set\ xs$ and $B = set\ ys$ are given by sorted and duplicate-free lists, we use the following function *cle* of type $\alpha\ option \Rightarrow \alpha\ list \Rightarrow \alpha\ list \Rightarrow bool$ – a similar function *lec* (not shown) deals with other case $A \sqsubseteq_1 \overline{B}$:

$$\begin{array}{lll} cle\ b & [] & [] = \neg\, pi^\alpha\ b\ None \\ cle\ b\ (x \cdot xs) & [] & = \neg\, pi^\alpha\ b\ \lfloor x \rfloor \wedge cle\ \lfloor x \rfloor\ xs\ [] \\ cle\ b & [] & (y \cdot ys) = \neg\, pi^\alpha\ b\ \lfloor y \rfloor \wedge cle\ \lfloor y \rfloor\ []\ ys \\ cle\ b\ (x \cdot xs) & (y \cdot ys) & = (if\ x < y\ then\ \neg\, pi^\alpha\ b\ \lfloor x \rfloor \wedge cle\ \lfloor x \rfloor\ xs\ (y \cdot ys) \\ & & \quad else\ if\ y < x\ then\ \neg\, pi^\alpha\ b\ \lfloor y \rfloor \wedge cle\ \lfloor y \rfloor\ (x \cdot xs)\ ys \\ & & \quad else\ \neg\, pi^\alpha\ b\ \lfloor x \rfloor) \end{array} \tag{7}$$

The additional parameter b acts as a lower bound: *cle* b interprets the complement $\overline{set\ xs}$ with respect to the set of values greater than b (notation $b\uparrow$) instead of *UNIV*. As it further assumes $set\ xs \cup set\ ys \subseteq b\uparrow$, it ignores all values not in $b\uparrow$. For example, *cle* $\lfloor 0 \rfloor$ treats the type from (5) as if it were $1 < 2$, i.e., it considers only the left half of (5).

Lemma 2. *Let α be finite, and xs and ys be sorted and duplicate-free, and $set\ xs \cup set\ ys \subseteq b\uparrow$. Then, $\overline{set\ xs} \cap b\uparrow \sqsubseteq set\ ys$ iff cle b xs ys.*

Corollary 2. *If α is finite, xs and ys are sorted and duplicate-free, then*

$$\overline{set\ xs} \sqsubseteq set\ ys = cle\ None\ xs\ ys.$$

Let us see how *cle* works. The first two cases correspond to $\overline{A} \cap b\uparrow \sqsubseteq \emptyset$ for $A = set\ []$ or $A = set\ (x \cdot xs)$. By P1, this holds iff $\overline{A} \cap b\uparrow = \emptyset$ – and the two equations use pi^α to test if A exhausts $b\uparrow$. The third case is symmetric: $b\uparrow \sqsubseteq set\ (y \cdot ys)$ holds iff $b\uparrow \subseteq set\ (y \cdot ys)$, i.e., $set\ (y \cdot ys)$ exhausts $b\uparrow$. The last case is the most interesting one. Suppose that $x < y$. Then, there must not be a value between b and x; otherwise $(pi^\alpha\ b\ \lfloor x \rfloor)$, the minimum element of $\overline{A} \cap b\uparrow$ is lower than the minimum element y of B, so $\overline{A} \cap b\uparrow \sqsupseteq_1 B$ by Lem. 1. Moreover, neither $y \cdot ys$ nor the complement of $x \cdot xs$ contain x, as the lists are sorted and duplicate-free. Thus, $\overline{set\ (x \cdot xs)} \cap b\uparrow = \overline{set\ xs} \cap \lfloor x \rfloor\uparrow$ and *cle* recurses. Now

suppose that $y < x$. Then, there must not be a value between b and y; otherwise $Min\ (\overline{A} \cap b\uparrow) < Min\ B$ and thus $B \sqsubset_1 \overline{A} \cap b\uparrow$ by Lem. 1. As $y < x$ and the lists are sorted, y is the minimum element of both $\overline{A} \cap b\uparrow$ and B. Thus, we can remove y from both sets in the recursive call by raising b to $\lfloor y \rfloor$ and dropping y from B. This is correct by Lem. 1. Finally, if $x = y$, we have found an element in which the sets differ. If $\neg\, pi\ b\ \lfloor x \rfloor$, then $y \in B$ is the minimum element of the symmetric difference between $\overline{A} \cap b\uparrow$ and B, so $B \sqsubset \overline{A} \cap b\uparrow$. Otherwise, the converse holds.

As can be seen from *cle*'s definition, every case requires at most two comparisons and one call to *pi*, and every recursive call consumes one list element. Thus, deciding $\overline{A} \sqsubseteq B$ is linear in the size of A and B if α is finite – for infinite α, this takes constant time by P4. To decide whether α is infinite, we use another type class to overload FIN^α with the meaning of *finite UNIV*. Finite types implement FIN^α as *True*, infinite ones as *False*.

In summary, (8) below implements the total order on sets, where ... represents the usual test for $corder^\alpha$ being defined and that A and B are finite (except for the first equation). The function *s2l A* returns A's elements as a sorted (w.r.t. $corder^\alpha$) and duplicate-free list – for $A = RSet\ rs$, *s2l rs* traverses rs in-order; for *DSet* (*MSet*), *s2l* sorts the elements (and removes duplicates); it fails with an exception for *ChF* and *Compl* as expected. The element type α must instantiate the type classes *corder*, *pi* and *FIN*. Note how (8) exploits that pattern matching is sequential (cf. §2.4): the last equation, e.g., executes only if A and B are no complements. Thus, sequentiality saves us from manually implementing all 28 cases.

$$\begin{array}{lll} Compl\ A \sqsubseteq Compl\ B & = \ldots & B \sqsubseteq A \\ Compl\ A \sqsubseteq B & = \ldots & FIN^\alpha \wedge cle\ None\ (s2l\ A)\ (s2l\ B) \\ A \sqsubseteq Compl\ B & = \ldots & FIN^\alpha \longrightarrow lec\ None\ (s2l\ A)\ (s2l\ B) \\ A \sqsubseteq B & = \ldots & lexord\ (>)\ (s2l\ A)\ (s2l\ B) \end{array} \tag{8}$$

3.3 Nesting and Extensibility

Still, we cannot use RBTs for sets of sets of sets, as we have not yet instantiated *pi* for sets. To implement $pi^{\alpha\ set}$, we must also know α's cardinality – to that end, we use the overloaded constant $card\text{-}UNIV^\alpha$ from [12]. To see why cardinality matters, consider the sets $\emptyset$ and $UNIV = \overline{\emptyset}$. Now, $pi^{\alpha\ set}\ \lfloor\emptyset\rfloor\ \lfloor\overline{\emptyset}\rfloor$ holds iff there is a set A with $\emptyset \sqsubset A \sqsubset UNIV$ iff more than one value inhabits α. Yet, pi^α does not suffice to decide this: As we represent $UNIV$ as $\overline{\emptyset}$, we do not get hold of any value of α, which we need for calling pi^α.

We now implement $pi^{\alpha\ set}$. The border cases are easy thanks to P1:

$$pi^{\alpha\ set}\ None\ \lfloor B \rfloor = (B \neq \emptyset) \qquad pi^{\alpha\ set}\ \lfloor A \rfloor\ None = (A \neq UNIV)$$

However, we can compute $pi^{\alpha\ set}\ \lfloor A \rfloor\ \lfloor B \rfloor$ only if A and B are (co)finite, i.e., we need to pattern-match on the pseudo-constructors like in (8). Note that $pi^{\alpha\ set}\ \lfloor Compl\ A \rfloor\ \lfloor Compl\ B \rfloor = pi^{\alpha\ set}\ \lfloor B \rfloor\ \lfloor A \rfloor$ holds by P3. For the other cases, we define auxiliary functions *PI*, *PIc*, and *cPI*. For example,

$$pi^{\alpha\ set}\ \lfloor A \rfloor\ \lfloor Compl\ B \rfloor = \ldots\ PIc\ None\ 0\ (s2l\ A)\ (s2l\ B) \tag{9}$$

where *PIc* satisfies (10) if α is finite, xs and ys are sorted and duplicate-free lists, and $set\ xs \cup set\ ys \subseteq b\uparrow$; $||A||$ denotes the number of elements of the set A.

$$PIc\ b\ ||UNIV - b\uparrow||\ xs\ ys = (\exists A \subseteq b\uparrow.\ set\ xs \sqsubset_1 A \wedge A \sqsubset_1 \overline{set\ ys} \cap b\uparrow) \quad (10)$$

Like *cle*, the three functions traverse the list representations and call pi^α at most linearly many times. Their definitions are technical, but provide no new insights.

At last, we can order arbitrary nestings of sets and thus implement them as RBTs efficiently. Yet, the specification (6) for pi^α violates our rule of making sure that every type can be instantiated: They depend on the default linear order $<$, which does not work for $\alpha\ set$. Therefore, we introduce another overloaded constant cpi^α – its specification is the same as (6) except that *corder* replaces $<$ and $corder^\alpha \neq None$ and FIN^α guard the equations. Hence, we have to implement cpi^α sensibly only for finite types α, for which we have provided an order, too. As most types are infinite, the restriction to finite types saves a lot of work.

4 Evaluation

To evaluate the efficiency and usability of our approach, we have performed two micro-benchmarks (§4.1 and §4.2) and integrated it with the Java interpreter (§4.3). All run-time tests ran on a Pentium DualCore E5300 2.6GHz with 2GB of RAM using Ubuntu GNU/Linux 9.10 and PolyML 5.4.1 or mlton 20100608; the figures are the average of four runs.

In preliminary tests, we noticed that Isabelle's default implementation based on lists sometimes outperformed LC with RBTs, even for large sets. We found that intermediate lists caused the slowdown. When we represent the sets as RBTs, *s2l* in code equations such as (8) and (9) first converts them into lists. While this simplifies the definitions and proofs, constructing the whole intermediate list is costly at run time - especially as the first few elements often suffice. Thus, we have manually eliminated such intermediate lists using the *destroy/unfoldr* pattern from shortcut fusion [18] before performing the benchmarks; we have proved the transformation correct in Isabelle.

4.1 Comparison with Other Approaches

The first micro-benchmark compares our approach with Isabelle's default implementation for sets, the ICF [9], and a conventional, RBT-based implementation. We start with the empty set, insert n numbers, then remove n numbers, then test n numbers for membership, and iterate over the set counting those elements less than n. All numbers are chosen randomly between 0 and $2n$ and implemented with ML's arbitrary precision integers. As discussed in [9], this benchmark measures the efficiency of the most common operations insertion, removal, membership and iteration, which all of the above implementations support.

Table 1 shows the run times under mlton for different n. As the first three rows all use RBTs, we can estimate the overhead that our approach (LC) and ICF add to a direct implementation with RBTs. LC's overhead is less than 1%,

Table 1. mlton run times in seconds for the comparison benchmark; the last column bounds standard deviation (in percent of the run times) over all n

Impl. / n	10k	20k	30k	40k	50k	100k	500k	1M	1.5M	2M	std. dev.
LC	.065	.138	.213	.295	.375	.825	5.17	10.8	17.0	23.3	$< 2.0\,\%$
ICF	.066	.139	.217	.300	.387	.839	5.17	11.8	17.4	23.9	$< 1.4\,\%$
RBT	.065	.135	.211	.292	.376	.818	5.59	10.8	17.0	23.3	$< 2.3\,\%$
default	1.50	5.72	3.08	22.8	36.2						<10 %

the ICF's varies between 2% and 10%. The last row refers to Isabelle's (much slower) default setup with lists, which has quadratic complexity. Under PolyML, the RBT-based implementations take more than twice as long, and the average overheads are 2.5% for both ICF and LC. In conclusion, our approach is as efficient as the ICF and would be a good replacement for Isabelle's default setup.

4.2 Nested Sets

This benchmark exercises the linear order on sets from §3. It starts with $\emptyset$ and inserts n sets to obtain a set $\mathcal{A}$ of sets. We generate each member set by inserting a random number of random numbers and randomly taking the complement; random numbers are chosen between 0 and m. Then, we check whether $\mathcal{A}$ contains another 100 sets generated the same way, and compute the size of the union of all sets in $\mathcal{A}$. We now use unsigned 32-bit words from Isabelle's word library for the numbers; so we exercise the *pi* implementation, too, as the word type is finite.

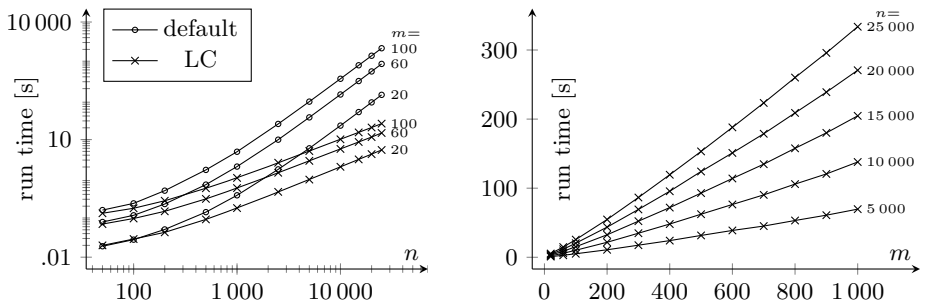


Fig. 1. mlton run times for the benchmark with nested sets

Figure 1 shows the mlton run times for Isabelle's default setup —○— and ours —×— – like in §4.1, PolyML runs take twice as long. In the log-log plot on the left, m is fixed and n varies. From $n = 1000$ on, the plots are linear with slope 2 and 1, respectively. As the slope denotes the exponent of the polynomial complexity, this confirms that Isabelle's setup is quadratic and ours almost linear. While

default is almost as fast as LC for small sets, LC is much faster for larger n and m. For $n = 25000$ and $m = 100$, e.g., it is 5.6 min vs. 35.6 min.

On the right, we now vary the size m of the inner sets for fixed numbers n of sets, but show only our approach LC. As the scales are linear, the plots fit a $m \log m$ curve. This shows that our linear order on α *set* is indeed efficient.

4.3 Case Study: Java Interpreter

In [15], we generated an interpreter for multithreaded Java from JinjaThreads, a large (86kLoC) Isabelle formalisation. Already for small programs, we achieved performance gains of 13% by pre-computing the lookup functions that extract information from the program declaration. We cached the information in (associative) lists using data refinement from α *set* and (α, β) *mapping*, Isabelle's type for finite executable maps. Type class restrictions disallowed more efficient implementations like RBTs, because the keys are (class) names, i.e., lists of chars, and ordering for lists has already been fixed to the (partial) prefix order.

Switching to our new approach was straightforward: we only had to import our new Isabelle files and instantiate the new type classes *ceq*, *corder*, and *set-impl* for the custom types as explained in §2. All existing definitions and proofs remained untouched. All in all, it took less than two hours. This shows that our approach is indeed easy to use.

To assess the impact on run-time, we took a Java program with 99 classes, converted it to JinjaThreads input syntax using Java2Jinja [13, §6.5], and ran the well-formedness checker (WF), the source code interpreter (SC), and the virtual machine (VM) on it. Table 2 shows the timings; the first row gives the timing for the original JinjaThreads versions without caching, the second with list-based caching. LC denotes our new approach: it gains 30% over the list-based implementation for WF, which heavily exercises the lookup functions and profits most from caching. As the interpreter calls the lookup functions less frequently, the gain is much smaller (1%). Surprisingly at first, LC ruins the VM performance – it is even slower than without caching. The bottleneck is the automatic selection of the set implementation. Internally, the VM uses sets also as a non-determinism monad the type of whose elements is built from 84 type constructors and 3 type variables. Hence, the execution of each bytecode instruction needs to query the available operations of all these constructors before it picks a set implementation. As a remedy, we disabled the automatic selection locally by replacing $\emptyset$ with *MSet* [] in the VM's code equations. This improves the runtime to .068 s and requires only three Isabelle declarations. To achieve the same performance as list-based tabulation (.053 s), we further replace the other operations on these sets with those that only have code equations for *MSet*. Hence, we save the dictionary constructions that emulate the type classes *ceq* and *corder* in ML.

Table 2. PolyML run times [s] for the Java interpreter

	WF	SC	VM
w/o	2.51	5.81	.086
lists	.013	5.39	.053
LC	.009	5.35	.106 (.053)

5 Conclusion and Future Work

We have proposed a light-weight approach to getting efficient code from Isabelle formalisations based on type classes and code generator refinement. It is flexible, extensible, and nestable. Our benchmarks and a case study show that it is indeed efficient, easy to use, and fits in the existing Isabelle libraries.

Four features of Isabelle's code generator have been crucial for this work:

Incremental declarations are the key to extensibility, as we can adjust the code generator setup right until code generation. This allows us, e.g., to bypass the impossibility results for single-parameter type classes [4,17].

Data refinement permits to represent values differently in logic and code. For configuration options for code generation, we suggest to take this to extremes (as shown in §2.3): Merge all cases in the logic! So, users can add further cases (data refinement) and change the configuration (program refinement).

Type classes enable overloading, our generated code uses them to query polymorphic type parameters. Type classes for code generation should be independent of those for logical concepts (e.g., *corder* vs. *linorder*, §2). For extensibility, they should be definitional such that every type can instantiate them.

Sequential pattern matching has helped to keep the effort linear in the number of implementations, see §2.4 and (8). This feature is hardly known; in Isabelle2013, such equations must be declared in the reversed order.

The next step is to cover more container types and implementations, e.g., bags and hash tables. Moreover, we want to integrate LC with Isabelle's packages such that they instantiate the type classes automatically. Also, we hope to equip Isabelle's code generator with an analysis that catches exceptions due to unsupported operations already at generation time. Future case studies will show how much effort LC saves in new developments.

Acknowledgements. We thank D. Lohner and J. Breitner for valuable discussions and comments on earlier drafts and F. Haftmann and L. Bulwahn for helping with the code generator. This work has been partially funded by DFG grant Sn11/10-2.

References

1. Appel, A.W.: Efficient verified red-black trees (2011), `http://www.cs.princeton.edu/~appel/papers/redblack.pdf`
2. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: SEFM 2004, pp. 230–239. IEEE Computer Society (2004)
3. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 147–163. Springer, Heidelberg (2009)
4. Chen, K., Hudak, P., Odersky, M.: Parametric type classes. In: LFP 1992, pp. 170–181. ACM (1992)

5. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Funct. Program. 18(1), 15–46 (2008)
6. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer, Heidelberg (2013)
7. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
8. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Sys. 28, 619–695 (2006)
9. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010)
10. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012)
11. Lescuyer, S.: Containers: a typeclass-based library of finite sets/maps (2011), `http://coq.inria.fr/pylons/contribs/view/Containers/v8.3`
12. Lochbihler, A.: Formalising FinFuns – generating code for functions as data from Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 310–326. Springer, Heidelberg (2009)
13. Lochbihler, A.: A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)
14. Lochbihler, A.: Light-weight containers. Archive of Formal Proofs, Formal proof development (2013) `http://afp.sf.net/entries/Containers.shtml`
15. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 216–232. Springer, Heidelberg (2011)
16. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
17. Peyton Jones, S.: Bulk types with class. In: Haskell Workshop 1997 (1997)
18. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: ICFP 2002, pp. 124–132. ACM (2002)
19. Thiemann, R.: Generating linear orders for datatypes. Archive of Formal Proofs, Formal proof development (2012), `http://afp.sf.net/entries/Datatype_Order_Generator.shtml`

Ordinals in HOL: Transfinite Arithmetic up to (and Beyond) ω_1

Michael Norrish[1] and Brian Huffman[2]

[1] Canberra Research Lab., NICTA*
also, Australian National University
Michael.Norrish@nicta.com.au
[2] Galois, Inc.
huffman@galois.com

Abstract. We describe a comprehensive HOL mechanisation of the theory of ordinal numbers, focusing on the basic arithmetic operations. Mechanised results include the existence of fixpoints such as ε_0, the existence of normal forms, and the validation of algorithms used in the ACL2 theorem-proving system.

1 Introduction

The ordinal numbers are an important foundational type in axiomatic set theory; used there, for example, in the definition of the von Neumann hierarchy and the cardinal numbers. In logic, ordinal numbers also provide an important characterisation of the strength of various logical systems.

Unfortunately, the typed logic implemented in the various HOL systems (including Isabelle/HOL) is not strong enough to define a type for all possible ordinal values (a proper class in a set theory like NBG). It turns out, however, that for any fixed $n \in \mathbb{N}$, we *can* model all ordinals of cardinality $\aleph_n$. The user is thus able to choose an ordinal domain of sufficient size for their purposes.

Our approach is to model ordinals as quotients of wellorders with respect to wellorder isomorphism. This approach has not been mechanised before. Within HOL, every wellorder has some underlying domain (represented as a polymorphic type argument). The resulting ordinals are also parameterised by a type argument, indirectly encoding the limit of the type. For example, the type *num ordinal* captures only the countable ordinals.

One important use of ordinal numbers occurs in the ACL2 theorem-proving system, which uses ordinal numbers as part of its termination reasoning for recursive definitions. Recently, Manolios and Vroon [6] improved ACL2's representation of ordinal numbers, and implemented new, more efficient algorithms for manipulating those numbers. Their work mechanised proofs of the correspondence between the old and new notational systems, and also proved the expected arithmetic properties. However, ACL2 has no

* 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. The first author also thanks Thomas Forster for productive early discussions. Finally, we thank ITP's anonymous reviewers for helpful comments.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 133–146, 2013.

notation-independent theory of the ordinals available to it, and so no way to model the set-theoretic ordinals. In *this* paper, we provide a sufficiently rich model, and are thus able to validate Manolios and Vroon's algorithms.

Contribution. This work makes a particular contribution in

- its definition of ordinal supremum,
- the fact that the mechanised notion of ordinal is polymorphic in an underlying universe type (allowing ordinals of large cardinality), and
- its mechanised validation of the ACL2 algorithms for ordinal arithmetic

HOL4 Notation and Theorems. All statements appearing with a turnstile ($\vdash$) are HOL4 theorems, automatically pretty-printed to LaTeX. Notation specific to this paper is explained as it is introduced. Otherwise, HOL4's syntax is a generally pleasant combination of quantifiers ($\forall$, $\exists$) and functional programming.

The option type α *option*, often used to encode partial functions, includes values NONE, and SOME x for all possible values x of type α. Hilbert choice is available through the epsilon notation. Read $\varepsilon x.\ P\ x$ as "the x that satisfies (predicate) P".

Sets and characteristic functions (of type $\alpha \to$ *bool* for element type α) are identified. Sets support standard operations such as union ($\cup$), and element removal (s DELETE e). The term BIGUNION s denotes the union of a set of sets. We write f ' s for the image of the set s under function f. BIJ $f\ s_1\ s_2$ means that function f is a bijection between sets s_1 and s_2. The universal set over type α is written $\mathcal{U}(:\alpha)$. Cardinality reasoning is expressed with $s \preccurlyeq t$ ("there is an injection from s to t"), and $s \approx t$ ("there is a bijection between s and t").

2 Wellorders

Definition 1. *We define what it is for a relation R to be a wellorder:*

$$\begin{aligned}&\mathsf{wellorder}\ R \iff \\ &\quad \mathsf{wellfounded}\,(\mathsf{strict}\ R)\ \wedge \\ &\quad \mathsf{linear\_order}\ R\,(\mathsf{domain}\ R\ \cup\ \mathsf{range}\ R)\ \wedge \\ &\quad \mathsf{reflexive}\ R\,(\mathsf{domain}\ R\ \cup\ \mathsf{range}\ R)\end{aligned}$$

As there is at least one value satisfying this definition (the empty set will do), we use HOL's standard type definition mechanism to define a new type (family) α *wellorder* that captures all of the possible wellorders over values drawn from arbitrary types α. The critical relations over wellorders are order-isomorphism and the relation that orders them linearly. The first is straightforward.

Definition 2. *Two wellorders are isomorphic if there is a bijective function (conjuncts two and three below) between their respective fields (conjunct one) that preserves the ordering (conjunct four):*

$$
\begin{array}{l}
w_1 \approx_w w_2 \iff \\
\quad \exists f. \\
\quad\quad (\forall x.\ x \in \mathsf{fld}\, w_1 \Rightarrow f\ x \in \mathsf{fld}\, w_2) \wedge \\
\quad\quad (\forall x_1\ x_2. \\
\quad\quad\quad x_1 \in \mathsf{fld}\, w_1 \wedge x_2 \in \mathsf{fld}\, w_1 \Rightarrow \\
\quad\quad\quad\quad (f\ x_1 = f\ x_2 \iff x_1 = x_2)) \wedge \\
\quad\quad (\forall y.\ y \in \mathsf{fld}\, w_2 \Rightarrow \exists x.\ x \in \mathsf{fld}\, w_1 \wedge f\ x = y) \wedge \\
\quad\quad \forall x\ y.\ (x, y) \in {w_1}^{\neq} \Rightarrow (f\ x, f\ y) \in {w_2}^{\neq}
\end{array}
$$

(We are using syntax overloading to simplify notation: the formula $(x, y) \in {w_1}^{\neq}$ *means that the pair* (x, y) *is a strict inequality in the relation (a set of pairs) that represents the wellorder value* w_1*. Alternatively, read* $(x, y) \in {w_1}^{\neq}$ *as "*x *is strictly less than* y *in* w_1*". Read* $\mathsf{fld}\, w$ *as the union of the domain and range of the relation representing* w*.)*

The definition of the ordering relation on wellorders depends on the wobound function, which truncates a wellorder so that it includes only those elements below a particular point. There are two important theorems about wobound:

$$
\begin{array}{l}
\vdash\ (x, y) \in (\mathsf{wobound}\, z\ w)^{\neq} \iff \\
\qquad (x, z) \in w^{\neq} \wedge (y, z) \in w^{\neq} \wedge (x, y) \in w^{\neq} \\
\vdash\ (x, y) \in w^{\neq} \Rightarrow \mathsf{wobound}\, x\, (\mathsf{wobound}\, y\ w) = \mathsf{wobound}\, x\ w
\end{array}
$$

Definition 3. *The ordering relation for wellorders (written* $w_1 \prec_w w_2$*) can then be defined*

$$w_1 \prec_w w_2 \iff \exists x.\ x \in \mathsf{fld}\, w_2 \wedge w_1 \approx_w \mathsf{wobound}\, x\ w_2$$

Transitivity of $\prec_w$ follows from the transitivity of order-isomorphism and the second result about wobound above. Well-foundedness for $\prec_w$ follows easily from the well-foundedness of the underlying relation. Well-foundedness is also the basis for the proof that $\prec_w$ is irreflexive. Finally, we show that $\prec_w$ is trichotomous:

Theorem 1.

$$\vdash\ w_1 \prec_w w_2 \vee w_1 \approx_w w_2 \vee w_2 \prec_w w_1$$

The proof of this result is the most involved of this section.

Proof. Let w_1 and w_2 be wellorders over α and β respectively. We define f of type $\alpha \to \beta\ option$ by well-founded recursion. The value of $f\ x$ is $\mathsf{SOME}\ y$ when y is the least element in w_2 not in the image of f applied to all elements less than x. If there is no such y, then $f\ x = \mathsf{NONE}$. If there is an x such that $f\ x = \mathsf{NONE}$, then w_2 is less than w_1, and the least value x where $f\ x = \mathsf{NONE}$ is the bound needed to demonstrate this. If f never has value NONE, then w_2 is at least as big as w_1. If the image of f on the elements of w_1 is all the elements of w_2, then f is the bijection we need to demonstrate order-isomorphism of w_1 and w_2. Otherwise, there is an element of w_2 not in the image of f. Take the least such element to be the bound demonstrating $w_1 \prec_w w_2$. □

3 Constructing the Ordinals

With $\approx_w$ an equivalence relation, we can quotient all possible wellorders over the type α, giving us a natural type of ordinals over α. However, if α is a finite type, then there are only finitely many ordinals over this type. Clearly, all the interesting ordinals are those over infinite types, and so our approach is to make the new type $\alpha\,\mathit{ordinal}$ a quotient over the wellorders over the sum type $\alpha + \mathit{num}$. Henceforth, this type is abbreviated as $\alpha\,\mathit{inf}$.

This construction means that the distinct types $\mathit{unit}\,\mathit{ordinal}$, $\mathit{bool}\,\mathit{ordinal}$ and $\mathit{num}\,\mathit{ordinal}$ will all be isomorphic (they will all be copies of the countable ordinals). On the other hand, the type $(\mathit{num} \to \mathit{bool})\,\mathit{ordinal}$ is large enough to include the first uncountable ordinal, ω_1.

When we quotient, and create the new type $\alpha\,\mathit{ordinal}$, the $(\prec_w)$ relation lifts to the new type, defining $(<)$. This relation inherits the irreflexivity, transitivity, well-foundedness and trichotomy results of $(\prec_w)$. Using these, it is trivial to show that the ordinals themselves form a well-order.

Definition 4. *Well-foundedness also allows the definition of a "least" operator for ordinals:*

$$(\mathsf{oleast})\,(P : \alpha\,\mathit{ordinal} \to \mathit{bool}) = \\ \varepsilon(x : \alpha\,\mathit{ordinal}).\, P\,x \;\wedge\; \forall(y : \alpha\,\mathit{ordinal}).\, y < x \;\Rightarrow\; \neg P\,y$$

This is well-defined as long as the predicate (or set) P is not everywhere false (the empty set).

Syntactically, we make (oleast) a binder, allowing us to write terms such as $(\mathsf{oleast}\,x.\, y < x)$ (the definition of the successor of y, written y^+), and $(\mathsf{oleast}\,x.\,\mathsf{T})$ (the zero-ordinal).

This copy of the natural numbers is a good starting point. It is straightforward to inject HOL's natural numbers with a new constant: $\&:\mathit{num} \to \alpha\,\mathit{ordinal}$, which is also the basis for ordinal numerals (0, 1, 2 *etc*).

Definition 5. *Write* $\mathsf{preds}\,\alpha$ *to denote the set of all predecessors of an ordinal.*

Definition 6. *Define the notion of a set being downward closed:*

$$\vdash \mathsf{downward\_closed}\,s \iff \forall a\, b.\, a \in s \wedge b < a \Rightarrow b \in s$$

Von Neumann famously characterised the ordinal numbers as those sets equal to their own predecessors. HOL's type system doesn't allow this: instead we must replace "equal" with the existence of a bijection:

Theorem 2. *The* preds *function forms a bijection between all possible ordinals and all but one of the downward closed sets of ordinals. The one omission is the universal set.*

$$\vdash \mathsf{BIJ}\ \mathsf{preds}\ \mathcal{U}(:\alpha\ \mathit{ordinal})\ (\mathsf{downward\_closed}\ \mathsf{DELETE}\ \mathcal{U}(:\alpha\ \mathit{ordinal}))$$

3.1 Cardinality Arguments and Supremum

We want a constant $(\mathsf{sup} : (\alpha\, \mathit{ordinal} \to \mathit{bool}) \to \alpha\, \mathit{ordinal})$, that takes a set of ordinals as an argument and returns their supremum. In our setting, this function can't be well-defined on all possible arguments: when passed the universal set of all α-ordinals, there is no possible value to return. Nonetheless, we can characterise those situations when $\mathsf{sup}\ s$ does have a reasonable value. First, the definition:

Definition 7. *The supremum of a set is the least element not in the set's collective predecessors.*

$$\mathsf{sup}\ oset = \mathsf{oleast}\, \alpha.\ \alpha\ \notin\ \mathsf{BIGUNION}\,(\mathsf{preds}\ `\ oset)$$

To characterise the reasonable arguments to sup, we need a definition and two further theorems.

Definition 8. *Let* allOrds *be the wellorder of all the α-ordinals (ordered by $(<)$).*

$$(\mathsf{allOrds} : \alpha\, \mathit{ordinal}\, \mathit{wellorder}) = \mathsf{mkWO}\,\{(x, y)\ |\ x = y\ \vee\ x < y\}$$

(The constant mkWO *lifts a relation into the type α wellorder. It is necessary to separately show that the relation satisfies the* wellorder *predicate from Definition 1.)*

Theorem 3. *Any wellorder w over the type α inf is order-isomorphic to the segment of* allOrds *below the element of α ordinal to which the quotienting (*mkOrdinal*) maps w.*

$$\begin{array}{ll} \vdash & (w : \alpha\, \mathit{inf}\, \mathit{wellorder}) \approx_w \\ & \mathsf{wobound}\,(\mathsf{mkOrdinal}\ w)\ (\mathsf{allOrds} : \alpha\, \mathit{ordinal}\, \mathit{wellorder}) \end{array}$$

Note how w is a wellorder over α inf, but that the right-hand side of the isomorphism is a wellorder over all possible ordinals *over α inf.*

Proof. By contradiction. Then, by well-foundedness, there is a least w where the isomorphism doesn't hold. If the two wellorders are not order-isomorphic, one is smaller than the other, by trichotomy of $(\prec_w)$. If w is smaller, there is a bound b (an ordinal) in allOrds, smaller still than mkOrdinal w, such that

$$w \approx_w \mathsf{wobound}\ b\ \mathsf{allOrds}$$

There is a wellorder bw that is a member of b's equivalence class. As $(\prec_w)$ is reflected by $(<)$, we have $bw \prec_w w$. Because w was least, bw must be order-isomorphic to wobound b allOrds. So, bw and w are order-isomorphic to the same ordinal (wobound b allOrds), but $bw \prec_w w$, contradicting the irreflexivity of $(\prec_w)$. The other direction (when w is larger) is similar. □

This result means that the predecessors of any given α *ordinal* must be equinumerous to a wellorder over α *inf*.

Corollary 1. *The predecessors of any ordinal have cardinality no greater than that of (all of) the underlying set, α inf.*

$$\vdash\ \mathsf{preds}\ a \preccurlyeq \mathcal{U}(:\alpha\, \mathit{inf})$$

Theorem 4. *The cardinality of all of the type α ordinal is strictly greater than that of the type α inf.*

$$\vdash \mathcal{U}(:\alpha\ \mathit{inf}) \prec \mathcal{U}(:\alpha\ \mathit{ordinal})$$

Proof. By contradiction. If $\mathcal{U}(:\alpha\ \mathit{ordinal}) \preccurlyeq \mathcal{U}(:\alpha\ \mathit{inf})$, then the injection from left-to-right copies the wellorder allOrds into a wellorder wo of type α *inf wellorder*. This gives allOrds $\approx_w wo$. By Theorem 3, this wo is also order-isomorphic to wobound (mkOrdinal wo) allOrds. Transitivity of $(\approx_w)$ then gives us that allOrds is less than itself, which is impossible. □

These results then combine to give us the important characterising theorem about sup:

Theorem 5. *As long as the cardinality of the set s is not greater than that of $\mathcal{U}(:\alpha\ \mathit{inf})$, an arbitrary ordinal α is less than that set's supremum iff there is an element of s that is bigger than α.*

$$\begin{aligned}\vdash\ & s \preccurlyeq \mathcal{U}(:\alpha\ \mathit{inf}) \Rightarrow \\ & \forall\alpha.\ \alpha < \mathsf{sup}\ s \iff \exists\beta.\ \beta \in s \wedge \alpha < \beta\end{aligned}$$

Proof. sup takes the union of all the predecessors of all the elements of s (Definition 7). Let κ be the cardinality of $\mathcal{U}(:\alpha\ \mathit{inf})$. By Corollary 1 above, the predecessors of each element of set s have that cardinality. If s has no more than the same cardinality, then from the fact that $\kappa \times \kappa \approx \kappa$ and Theorem 3 above, the union calculated in the definition of sup cannot be the universal set of all possible ordinals. There must then be a least ordinal not within that union, and so sup s will be well-defined.

Moreover, the set of all the combined predecessors (call it ps) is also downward closed, and so, by Theorem 2, there must be an ordinal α whose predecessors are exactly ps. So, sup $s = \alpha$, and it is easy to show that the theorem's characterisation of its predecessors is correct. □

An easy corollary is that there is no maximal ordinal. For any ordinal α, we observe that $s = \mathsf{preds}\ \alpha \cup \{\alpha\}$ is downward closed and not equal to $\mathcal{U}(:\alpha\ \mathit{ordinal})$. Then, by Theorem 2 there must be a $\beta > \alpha$, with $s = \mathsf{preds}\ \beta$.

3.2 Limit Ordinals

Definition 9. *With* sup *defined, it is possible to define ω. This, the first limit ordinal, is the supremum of the copy of the natural numbers that injects into the ordinals* via (&).

$$(\omega : \alpha\ \mathit{ordinal}) = \mathsf{sup}\ \{((\&i) :\alpha\ \mathit{ordinal}) \mid \mathsf{T}\}$$

Definition 10. *We also define a constant* omax*, which returns the maximal element of a set of ordinals, if any. The option type is used to encode the partiality of this function, so the type of* omax *is $(\alpha\ \mathit{ordinal} \to \mathit{bool}) \to \alpha\ \mathit{ordinal}\ \mathit{option}$. If* omax (preds a) *is* NONE*, we abbreviate this condition as* islimit a.

Theorem 6. *One simple consequence of these definitions is that every natural number is less than ω, and that only the natural numbers are less than ω:*

$$\vdash (a : \alpha\ \mathit{ordinal}) < (\omega : \alpha\ \mathit{ordinal}) \iff \exists(n : \mathit{num}).\ a = ((\&n) :\alpha\ \mathit{ordinal})$$

4 Arithmetic

Theorem 7. *With access to a total well-founded relation* $(<)$*, we have always been able to define functions by well-founded recursion. However, we can now recast this in a more palatable form, one that makes the ordinals look a little like an algebraic type generated by constructors* 0*,* x^+ *and* sup s *(with* s *not including its own upper bound):*

$$\begin{array}{l}\vdash\ \forall(z:\beta)\,(sf:\alpha\,\mathit{ordinal}\to\beta\to\beta)\,(lf:\alpha\,\mathit{ordinal}\to(\beta\to\mathit{bool})\to\beta).\\ \quad\exists(h:\alpha\,\mathit{ordinal}\to\beta).\\ \quad\ h\,(0:\alpha\,\mathit{ordinal})=z\ \wedge\ (\forall(a:\alpha\,\mathit{ordinal}).\,h\,a^+=sf\ a\,(h\,a))\ \wedge\\ \quad\ \forall(a:\alpha\,\mathit{ordinal}).\\ \quad\quad(0:\alpha\,\mathit{ordinal})<a\ \wedge\ \mathsf{islimit}\,a\ \Rightarrow\ h\,a=lf\ a\,((h\ `\,(\mathsf{preds}\,a)):\beta\to\mathit{bool})\end{array}$$

This recursion theorem allows the user to specify three cases: z, a value in the desired range (β) for zero; sf, a function for constructing a result when h is passed a successor; and lf when the argument to h is a limit ordinal. The lf function is given the original limit ordinal a as well as the set of all the values given by recursive calls of h on a's predecessors.

The recursion theorem is all we need to define ordinal addition, multiplication and exponentiation. Working out the details for addition $(a+b)$: we will recurse on b, and let z be the value a, sf be $(\lambda x\ r.\ r^+)$, and lf be $(\lambda x\ rs.\,\mathsf{sup}\,rs)$. This gives

Definition 11. *Ordinal addition:*

$$\begin{array}{l}a+0=a\\ a+b^+=(a+b)^+\\ 0<b\ \wedge\ \mathsf{islimit}\,b\ \Rightarrow\ a+b=\mathsf{sup}\,((+)\,a\ `\,(\mathsf{preds}\,b))\end{array}$$

The definitions of multiplication and exponentiation are as straightforward. For example:

Definition 12. *Ordinal exponentiation:*

$$\begin{array}{l}a^0=1\\ a^{b^+}=a^b\cdot a\\ 0<b\ \wedge\ \mathsf{islimit}\,b\ \Rightarrow\ a^b=\mathsf{sup}\,((\mathsf{**})\,a\ `\,(\mathsf{preds}\,b))\end{array}$$

$((\mathsf{**})\,a)$ *is equivalent to* $(\lambda b.\ a^b)$*; the pretty-printing obscures this because the underlying constant prints as* $(\mathsf{**})$ *when it doesn't have two arguments.*

Reasoning about these operations is made easier by the observation that all three are *continuous* (in their second arguments). For addition, the continuity result is

$$\vdash\ s\preccurlyeq\mathcal{U}(:\alpha\,\mathit{inf})\ \wedge\ s\neq\emptyset\ \Rightarrow\ a+\mathsf{sup}\,s=\mathsf{sup}\,((+)\,a\ `\,s)$$

Rewriting with these theorems allows operators such as $(+)$ to move under sup arguments, where further simplification is usually possible. For example, the proofs (by induction) that addition and multiplication are associative are greatly simplified by their continuity theorems.

4.1 Division and Modulus

The various arithmetic operations on ordinal numbers do not satisfy many of the typical properties of number systems. For example, addition and multiplication are not commutative. However, they both enjoy cancellation properties for common arguments on the left:

$$\vdash \alpha + \beta = \alpha + \gamma \iff \beta = \gamma$$
$$\vdash \alpha \cdot \beta = \alpha \cdot \gamma \iff \alpha = 0 \vee \beta = \gamma$$

These then lead to the existence of unique quotients and remainders.

Theorem 8.

$$\vdash 0 < b \Rightarrow a = b \cdot (a \,/\, b) + a \bmod b \wedge a \bmod b < b$$
$$\vdash 0 < b \wedge a = b \cdot q + r \wedge r < b \Rightarrow q = a \,/\, b \wedge r = a \bmod b$$

Proof. The existence of the division and modulus constants is shown by taking the quotient d to be $\mathsf{sup}\{c \mid b \cdot c \leq a\}$. (The supremum is well-defined because the set is bounded above.) Then $b \cdot d \leq a$ follows from the continuity of multiplication. The existence of the remainder follows from an earlier result that $\vdash a \leq b \iff \exists c.\, b = a + c$.

The uniqueness result proceeds by first showing the uniqueness of the quotient (uniqueness of the modulus then follows from additive cancellation). If there is another quotient q' not equal to $a \,/\, b$ (write q), then it is either larger or smaller. If larger, then $q' = q + \delta$, for some non-zero δ, and $a = b(q + \delta) + r'$, where r' is the remainder accompanying q'. Then $a = bq + b\delta + r$, and cancellation and associativity then give us that $b\delta + r = a \bmod b$. But $0 < \delta$, making $a \bmod b$ too large. The other case is similar. □

4.2 Cantor Normal Forms

In a discrete domain such as the ordinals, division approximates multiplication's inverse, leaving a remainder. Analogously, with exponentiation we can construct a discrete logarithm. If working with base b, and e is the largest value such that b^e is under the target a, then we can "drop down" to the level of multiplication and find how many whole copies of b^e fit into a, giving us a c such that $b^e \cdot c \leq a$. Then we can repeat the process with the remainder.

Done over the natural numbers with $b = 10$, we derive a's decimal representation (strictly, the non-zero coefficients along with their indices). Over all ordinals, with $b = \omega$, we derive the *Cantor Normal Form* of a.

Definition 13. *The sequence of exponents and coefficients we derive in the above construction is the same as the information needed to specify a polynomial over a single variable. We define* eval_poly *to evaluate such sequences with respect to arbitrary bases:*

$$\mathsf{eval\_poly}\ b\ [\,] = 0$$
$$\mathsf{eval\_poly}\ b\ ((c, e) :: t) = b^e \cdot c + \mathsf{eval\_poly}\ b\ t$$

Definition 14. *We define* is_polyform *to capture well-formed "polynomial" sequences, requiring that the exponents are decreasing, and that the coefficients are always strictly between* 0 *and* b*:*

$$\begin{aligned}&\mathsf{is\_polyform}\ b\ [\,] \iff \mathsf{T}\\ &\mathsf{is\_polyform}\ b\ [(c, e)] \iff 0 < c \ \wedge\ c < b\\ &\mathsf{is\_polyform}\ b\ ((c_1, e_1) :: (c_2, e_2) :: t) \iff\\ &\quad 0 < c_1 \ \wedge\ c_1 < b \ \wedge\ e_2 < e_1 \ \wedge\ \mathsf{is\_polyform}\ b\ ((c_2, e_2) :: t)\end{aligned}$$

Then, just as with division, we are able to prove that "polynomial forms" always exist, and that they are unique.

Theorem 9. *For all ordinals* a*, and bases* b *greater than* 1*, it is possible to express* a *as the sum of a sequence of pairs of coefficients and powers-of-*b*. In the sequence each successive exponent is smaller than its predecessors.*

$$\vdash\ 1 < b\ \Rightarrow\ \exists ces.\ \mathsf{is\_polyform}\ b\ ces\ \wedge\ a = \mathsf{eval\_poly}\ b\ ces$$

Define the new constant polyform *to return such a sequence when given parameters* b *and* a *(if* $b < 2$*, allow that the function has no definite value). We show that all possible sequences with the desired property have* polyform*'s value:*

$$\begin{aligned}\vdash\ & 1 < b\ \wedge\ \mathsf{is\_polyform}\ b\ ces\ \wedge\ a = \mathsf{eval\_poly}\ b\ ces\ \Rightarrow\\ & \quad \mathsf{polyform}\ b\ a = ces\end{aligned}$$

Proof. Both proofs are by induction on the argument a. The first proof is similar to the proof of the existence of a quotient: the leading exponent is taken to be $\mathsf{sup}\,\{e \mid b^e \leq a\}$. After the coefficient c is calculated by division, and $b^e \cdot c$ subtracted out, the remaining ordinal is smaller and the inductive hypothesis applies. The uniqueness proof hinges on the following important lemma:

$$\vdash\ 1 < b\ \wedge\ \mathsf{is\_polyform}\ b\ ((c, e) :: t)\ \Rightarrow\ \mathsf{eval\_poly}\ b\ t < b^e$$

4.3 Fixpoints and ε_0

A function $(f\ :\ \alpha\,\mathit{ordinal} \to \alpha\,\mathit{ordinal})$ can be iterated any number of times from a starting value x. The resulting set $\{x, f(x), f(f(x)), \ldots, f^n(x), \ldots\}$ is clearly only countably infinite, and so will always have a supremum. Under certain conditions, that supremum will also be a fixpoint for f.

Theorem 10. *If* f *is non-decreasing and continuous, then it has a fixpoint. In fact, for* any *lower bound* a*, the function* f *has a fixpoint at least as large as* a*.*

$$\begin{aligned}\vdash\ & (\forall s.\ s \neq \emptyset \ \wedge\ s \preccurlyeq \mathcal{U}(:\alpha\,\mathit{inf}) \ \Rightarrow\ f\,(\mathsf{sup}\ s) = \mathsf{sup}\,(f\ `\ s)) \ \wedge\\ & (\forall x.\ x \leq f\ x) \ \Rightarrow\\ & \quad \forall a.\ \exists b.\ a \leq b \ \wedge\ f\ b = b\end{aligned}$$

Proof. Let s be the set above (all the f-iterates of a). Take the fixpoint to be the supremum of s. We have that $f\,(\mathsf{sup}\,s) = \mathsf{sup}\,(f\;`\;s)$, and are required to show that $\mathsf{sup}\,(f\;`\;s) = \mathsf{sup}\,s$. This follows straightforwardly because f is non-decreasing. □

Our arithmetic operations (addition, multiplication and exponentiation) are all continuous in their right arguments and non-decreasing. So, for example, the first fixpoint of $((\cdot)\,2)$ is 0; the next is ω, and the third is $\omega \cdot 2$. The first non-zero fixpoint of $((\cdot)\,\omega)$ is ω^ω. However, it turns out that the first fixpoint of $((^{**})\,\omega)$ is not expressible with any of the notation we have developed thus far.

Definition 15. *Let ε_0 be the least fixpoint of $((^{**})\,\omega)$:*

$$\varepsilon_0 = \mathsf{oleast}\,x.\,\omega^x = x$$

This is well-defined because of Theorem 10, giving us the following characterisations of ε_0:

$$\vdash\ \omega^{\varepsilon_0} = \varepsilon_0$$

$$\vdash\ a < \varepsilon_0 \Rightarrow a < \omega^a \wedge \omega^a < \varepsilon_0$$

Theorem 11. *As suggested, the arithmetic operations are all closed under ε_0:*

$$\vdash\ a < \varepsilon_0 \wedge b < \varepsilon_0 \Rightarrow a + b < \varepsilon_0 \qquad \vdash\ a < \varepsilon_0 \wedge b < \varepsilon_0 \Rightarrow a \cdot b < \varepsilon_0$$

$$\vdash\ a < \varepsilon_0 \wedge b < \varepsilon_0 \Rightarrow a^b < \varepsilon_0$$

5 Uncountable Ordinals

Definition 16. *The "countable ordinals" are those with countably many predecessors. Write* $\mathsf{countableOrd}\,a$ *for an a with this property.*

An immediate consequence of Theorem 4 is that there are uncountably many countable ordinals. To guarantee even larger ordinals, we must instantiate the α type-parameter with known-to-be-larger types:

Definition 17. *The α ucinf type has at least the cardinality of $2^{\aleph_0}$, and is thus at least as big as $\aleph_1$. The α ucord type contains ordinals that are quotients of wellorders over α ucinf:*

$$\alpha\,\mathit{ucinf} = (\alpha + (\mathit{num} \to \mathit{bool}))\,\mathit{inf}$$

$$\alpha\,\mathit{ucord} = (\alpha + (\mathit{num} \to \mathit{bool}))\,\mathit{ordinal}$$

Note that these abbreviations mean that every α ucord is also an ordinal. Every theorem about values of type α ordinal applies to values of type α ucord.

Lemma 1. *The countable ordinals are not larger than the universe of α ucinf (which contains $\mathcal{U}(:\mathit{num} \to \mathit{bool})$ as a subset).*

$$\vdash\ \{a \mid \mathsf{countableOrd}\,a\} \preccurlyeq \mathcal{U}(:\alpha\,\mathit{ucinf})$$

Proof. By contradiction. Then $\mathcal{U}(:\alpha\ \mathit{ucinf})$ injects into the countable ordinals *via* some f, but there is no bijection between the two. Let $a = \mathsf{sup}\,(f\ `\mathcal{U}(:\alpha\ \mathit{ucinf}))$. (The supremum is well-defined because the image cannot have greater cardinality than $\mathcal{U}(:\alpha\ \mathit{ucinf})$.) We now consider whether or not a is a countable ordinal.

If so, then we show that there is an injection from $\mathcal{U}(:\alpha\ \mathit{ucinf})$ into the (countable) predecessors of a, which gives an immediate contradiction. If the image of f doesn't includes the supremum, the injection is f itself. If there is a u such that $f\ u = a$, then f is an injection from $\mathcal{U}(:\alpha\ \mathit{ucinf})\ \mathsf{DELETE}\ u$ into the predecessors, and deleting a single element from an infinite set doesn't change its cardinality, so the contradiction can still be obtained.

If a is not a countable ordinal, then all of the countable ordinals must be among its predecessors. So, $\{b \mid \mathsf{countableOrd}\,b\} \preccurlyeq \mathsf{preds}\,a$. But we also have that $\mathsf{preds}\ a \preccurlyeq \mathcal{U}(:\alpha\ \mathit{ucinf})$, giving a contradiction by the transitivity of $(\preccurlyeq)$. □

Definition 18.

$$(\omega_1 : \alpha\ \mathit{ucord}) = \mathsf{sup}\,\{a \mid \mathsf{countableOrd}\,a\}$$

The supremum is well-defined because of Lemma 1 above.

Theorem 12. *The ordinal ω_1 is the first uncountable ordinal:*

$$\vdash\ x < \omega_1 \iff \mathsf{countableOrd}\,x$$

(The irreflexivity of $(<)$ means that ω_1 cannot itself be countable.)

6 Validating Algorithms on ACL2's Ordinals

The ACL2 system models ordinals up to ε_0 with a representation based on Cantor Normal Form. ACL2's manipulations of those values are defined by recursive functions over that syntax. ACL2 then takes as axiomatic that these recursive functions are correct; that, for example, its less-than relation on these values really does correspond to $(<)$.

In isolation, these axiomatic assertions can only be checked manually. However, thanks to work started by Gordon *et al.* [2], much of the ACL2 axiomatic system has been embedded in HOL4. More recently, the "ACL2 in HOL" project was completed by Kaufmann and Slind [5], who showed that ACL2's less-than relation is well-founded, justifying ACL2's recursion and induction principles. Kaufmann and Slind note in passing:

> *... we are not ascribing any semantics at all to the notation; a separate proof would be needed to show that indeed the following definitions do correspond to the ordinals up to ε_0.*

A little earlier, Manolios and Vroon [6] improved ACL's representation of the ordinals-up-to-ε_0, and developed efficient arithmetic algorithms for that representation. They noted:

Note that these proofs are not mechanically verified. To do so would require using a theorem prover that can reason both about ACL2 and set theory.

HOL4 can reason about ACL2, thanks to the embedding above, and can now also reason about ordinals in a way that captures their nature as canonical wellorders. Thus we are now in a position to do the mechanised proofs that could not be done in [5,6].

Kaufmann and Slind's HOL4 theory file defines the ACL2 ordinals to be

$$\mathit{osyntax} = \mathsf{End}\ \texttt{of}\ \mathit{num} \mid \mathsf{Plus}\ \texttt{of}\ \mathit{osyntax} \Rightarrow \mathit{num} \Rightarrow \mathit{osyntax}$$

That is, *osyntax* is an algebraic type with two constructors: End takes a natural number argument, and Plus takes a number and two *osyntax* values as arguments.

Definition 19. *The osyntax type can be given a semantics in α ordinal:*

$$\begin{aligned}[\![\mathsf{End}\ n]\!] &= \&n\\ [\![\mathsf{Plus}\ e\ c\ t]\!] &= \omega^{[\![e]\!]} \cdot \&c + [\![t]\!]\end{aligned}$$

Kaufmann and Slind define functions oless and is_ord, with the following equations for the interesting cases:

$$\begin{aligned}&\mathsf{oless}\ (\mathsf{Plus}\ e_1\ k_1\ t_1)\ (\mathsf{Plus}\ e_2\ k_2\ t_2) \iff\\ &\quad \texttt{if}\ \mathsf{oless}\ e_1\ e_2\ \texttt{then}\ \mathsf{T}\\ &\quad \texttt{else if}\ e_1 = e_2 \wedge k_1 < k_2\ \texttt{then}\ \mathsf{T}\\ &\quad \texttt{else if}\ e_1 = e_2 \wedge k_1 = k_2 \wedge \mathsf{oless}\ t_1\ t_2\ \texttt{then}\ \mathsf{T}\\ &\quad \texttt{else}\ \mathsf{F}\\ &\mathsf{is\_ord}\ (\mathsf{Plus}\ e\ k\ t) \iff\\ &\quad \mathsf{is\_ord}\ e \wedge e \neq \mathsf{End}\ 0 \wedge 0 < k \wedge \mathsf{is\_ord}\ t \wedge\\ &\quad \mathsf{oless}\ (\mathsf{expt}\ t)\ e\end{aligned}$$

The is_ord function is the analogue of is_polyform from Definition 14, capturing whether or not the notation is well-formed (non-zero coefficients and decreasing exponents). (The expt function returns e when applied to Plus $e\ c\ t$, and End 0 otherwise.)

Theorem 13. *The* oless *function is correct on well-formed osyntax values:*

$$\vdash \mathsf{is\_ord}\ x \wedge \mathsf{is\_ord}\ y \Rightarrow (\mathsf{oless}\ x\ y \iff [\![x]\!] < [\![y]\!])$$

And, ultimately:

Theorem 14. *The model function ($[\![\_]\!]$) is a bijection from well-formed osyntax values into the ordinals less than ε_0.*

$$\vdash \mathsf{BIJ}\ (\lambda x.\ [\![x]\!])\ \{x \mid \mathsf{is\_ord}\ x\}\ \{a \mid a < \varepsilon_0\}$$

6.1 Arithmetic

The ACL2 definitions of addition and multiplication over *osyntax* are correct:

Theorem 15.

$$\vdash \mathsf{is\_ord}\, x \wedge \mathsf{is\_ord}\, y \Rightarrow [\![\mathsf{ord\_add}\, x\, y]\!] = [\![x]\!] + [\![y]\!]$$

$$\vdash \mathsf{is\_ord}\, x \wedge \mathsf{is\_ord}\, y \Rightarrow [\![\mathsf{ord\_mult}\, x\, y]\!] = [\![x]\!] \cdot [\![y]\!]$$

Manolios and Vroon [6] note that ord_mult is very inefficient, and prove a version with better complexity, based on new constants pmult, c1 and c2. In order to embed it in ACL2, Manolios and Vroon have already mechanically proved pmult equivalent to ord_mult. Nonetheless, we also proved a version of their Theorem 10:

Theorem 16 (after Manolios and Vroon). *The efficient* pmult *algorithm correctly calculates ordinal multiplication. (The natural number parameter n can be set to zero initially.)*

$$\vdash \mathsf{is\_ord}\, a \wedge \mathsf{is\_ord}\, b \wedge n \leq \mathsf{c1}\, (\mathsf{expt}\, a)\, (\mathsf{expt}\, b) \Rightarrow [\![\mathsf{pmult}\, a\, b\, n]\!] = [\![a]\!] \cdot [\![b]\!]$$

7 Another Model

Sections 2 and 3 showed how to construct a type α *ordinal* using wellorders. In this section, we describe an alternative, earlier construction of ordinals that starts with the infinitely-branching tree datatype shown below.[1] The goal is to construct a wellordered type *ordinal* that has upper bounds for all countable sets; from this foundation, an Isabelle formalization by the second author [4] develops ordinal arithmetic as described in Section 4.

$$\mathit{preordinal} \;=\; \mathsf{Zero} \mid \mathsf{StrictLim}(\mathit{num} \to \mathit{preordinal})$$

We define the subterm relation $\lhd$ as the least transitive relation satisfying $f(x) \lhd \mathsf{StrictLim}(f)$ for all f, x. The wellfoundedness of $\lhd$ follows from the datatype induction rule for *preordinal*. However, $\lhd$ is not a wellorder because it is not linear. To construct a wellorder, we will need to quotient *preordinal* by a suitable equivalence relation.

We define relations $\preceq$ and $\prec$ as the smallest relations satisfying the following rules. Intuitively, $x \preceq y$ iff $(\preceq)$ relates every subterm of x to some subterm of y. We then define $x \approx y$ iff $x \preceq y \wedge y \preceq x$.

$$(\forall x.\; x \lhd y \Rightarrow x \prec z) \Rightarrow y \preceq z$$

$$x \preceq y \wedge y \lhd z \Rightarrow x \prec z$$

Wellfounded inductions (using $\lhd$) show that $\preceq$ is reflexive and transitive. It directly follows that $\approx$ is an equivalence relation. We can similarly prove further transitivity rules for various combinations of $\prec$ and $\preceq$. Finally we can prove the order trichotomy rule $x \prec y \vee y \prec x \vee x \approx y$ by nested inductions on x and y.

We then define type *ordinal* as a quotient $\mathit{preordinal}/{\approx}$. The various transitivity rules show that $\prec$ and $\preceq$ respect $\approx$, so we can lift them to relations $<$ and $\leq$ on the quotient type *ordinal*. The order trichotomy rule implies that *ordinal* is wellordered by $<$. Finally, we can construct a (strict) upper bound for any countably infinite set by lifting the constructor function StrictLim to type *ordinal*.

[1] If we replaced *num* with α *inf* in this data type, we might (this has not been pursued) then capture uncountable ordinals as in Section 5. Similarly, it is also plausible that a supremum constant akin to that in the model of Sections 2 and 3 should be definable for this type.

8 Conclusion

The HOL4 theory of ordinals demonstrates that ordinals can be cleanly modelled as a quotient of wellorders, that this approach supports ordinals of large cardinalities, and that supremum can be modelled as a function taking a set as an argument. All these contributions are novel with this work. In addition, the utility of the approach has been demonstrated by validating practically important algorithms in the ACL2 theorem-prover.

Related Work. There is relatively little published work on mechanisations of ordinals within a non-set-theoretic setting. One early development is John Harrison's `wellorder` library [3]. Originally developed for HOL88, this remains part of the HOL Light library. This theory picks out certain wellorders to be ordinals using Hilbert-choice, and proves some consequences of the Axiom of Choice, such as Zorn's Lemma. It does not include any ordinal arithmetic.

In similar vein, there is a large theory of ordinals and cardinals behind Traytel *et al.* [7]. This work is available at Isabelle's Archive of Formal Proofs. It defines wellorders and develops a number of important facts about cardinalities. It does not quotient its wellorder type, and emulates ordinal arithmetic "synthetically" (*e.g.*, addition as wellorder concatenation). This work does not define ordinal multiplication nor exponentiation.

Finally, as in ACL2, it is possible to use ordinal *notations* (capturing countably many ordinals). A great deal of interesting ordinal theory (up to Γ_0) has been mechanised in this style in Coq by Castéran and Contejean [1].

Availability. Most of the HOL4 theory of the ordinals described here is in the current release of HOL4. Newer material, including the validation of the ACL2 algorithms, was in the HOL4 repository by the time of commit `e3bd872ec1` and will appear in the next release. The sources for this paper are at `github.com/mn200/ordinals-paper`.

References

1. Castéran, P., Contejean, E.: On ordinal notations, `http://coq.inria.fr/V8.2pl1/contribs/Cantor.html`
2. Gordon, M.J.C., Reynolds, J., Hunt, Jr., W.A., Kaufmann, M.: An integration of HOL and ACL2. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD), pp. 153–160. IEEE Computer Society (2006)
3. Harrison, J.: The HOL wellorder library. HOL88 documentation (May 1992)
4. Huffman, B.: Countable ordinals. Archive of Formal Proofs, Formal proof development (November 2005), `http://afp.sf.net/entries/Ordinal.shtml`
5. Kaufmann, M., Slind, K.: Proof pearl: Wellfounded induction on the ordinals up to ε_0. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 294–301. Springer, Heidelberg (2007)
6. Manolios, P., Vroon, D.: Ordinal arithmetic: Algorithms and mechanization. Journal of Automated Reasoning 34(4), 387–423 (2005)
7. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, pp. 596–605. IEEE (2012)

Mechanising Turing Machines and Computability Theory in Isabelle/HOL

Jian Xu[1], Xingyuan Zhang[1], and Christian Urban[2]

[1] PLA University of Science and Technology, China
[2] King's College London, UK

Abstract. We formalise results from computability theory in the theorem prover Isabelle/HOL. Following the textbook by Boolos et al, we formalise Turing machines and relate them to abacus machines and recursive functions. We "tie the know" between these three computational models by formalising a universal function and obtaining from it a universal Turing machine by our verified translation from recursive functions to abacus programs and from abacus programs to Turing machine programs. Hoare-style reasoning techniques allow us to reason about concrete Turing machine and abacus programs.

1 Introduction

We like to enable Isabelle/HOL users to reason about computability theory. Reasoning about decidability of predicates, for example, is not as straightforward as one might think in Isabelle/HOL and other HOL theorem provers, since they are based on classical logic where the law of excluded middle ensures that $P \vee \neg P$ is always provable no matter whether the predicate P is constructed by computable means.

Norrish formalised computability theory in HOL4. He choose the λ-calculus as the starting point for his formalisation because of its "simplicity" [7, Page 297]. Part of his formalisation is a clever infrastructure for reducing λ-terms. He also established the computational equivalence between the λ-calculus and recursive functions. Nevertheless he concluded that it would be appealing to have formalisations for more operational models of computations, such as Turing machines or register machines. One reason is that many proofs in the literature use them. He noted however that [7, Page 310]:

> *"If register machines are unappealing because of their general fiddliness, Turing machines are an even more daunting prospect."*

In this paper we take on this daunting prospect and provide a formalisation of Turing machines, as well as abacus machines (a kind of register machines) and recursive functions. To see the difficulties involved with this work, one has to understand that Turing machine programs (similarly abacus programs) can be completely *unstructured*, behaving similar to Basic programs containing the infamous goto [3]. This precludes in the general case a compositional Hoare-style reasoning about Turing programs. We provide such Hoare-rules for when it *is* possible to reason in a compositional manner (which is fortunately quite often), but also tackle the more complicated case when we translate

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 147–162, 2013.

abacus programs into Turing programs. This reasoning about concrete Turing machine programs is usually left out in the informal literature, e.g. [2].

We are not the first who formalised Turing machines: we are aware of the work by Asperti and Ricciotti [1]. They describe a complete formalisation of Turing machines in the Matita theorem prover, including a universal Turing machine. However, they do *not* formalise the undecidability of the halting problem since their main focus is complexity, rather than computability theory. They also report that the informal proofs from which they started are not "sufficiently accurate to be directly usable as a guideline for formalization" [1, Page 2]. For our formalisation we follow mainly the proofs from the textbook by Boolos et al [2] and found that the description there is quite detailed. Some details are left out however: for example, constructing the *copy Turing machine* is left as an exercise to the reader—a corresponding correctness proof is not mentioned at all; also [2] only shows how the universal Turing machine is constructed for Turing machines computing unary functions. We had to figure out a way to generalise this result to n-ary functions. Similarly, when compiling recursive functions to abacus machines, the textbook again only shows how it can be done for 2- and 3-ary functions, but in the formalisation we need arbitrary functions. But the general ideas for how to do this are clear enough in [2].

The main difference between our formalisation and the one by Asperti and Ricciotti is that their universal Turing machine uses a different alphabet than the machines it simulates. They write [1, Page 23]:

> *"In particular, the fact that the universal machine operates with a different alphabet with respect to the machines it simulates is annoying."*

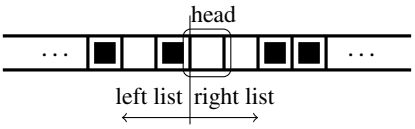
In this paper we follow the approach by Boolos et al [2], which goes back to Post [8], where all Turing machines operate on tapes that contain only *blank* or *occupied* cells. Traditionally the content of a cell can be any character from a finite alphabet. Although computationally equivalent, the more restrictive notion of Turing machines in [2] makes the reasoning more uniform. In addition some proofs *about* Turing machines are simpler. The reason is that one often needs to encode Turing machines—consequently if the Turing machines are simpler, then the coding functions are simpler too. Unfortunately, the restrictiveness also makes it harder to design programs for these Turing machines. In order to construct a universal Turing machine we therefore do not follow [1], instead follow the proof in [2] by translating abacus machines to Turing machines and in turn recursive functions to abacus machines. The universal Turing machine can then be constructed by translating from a (universal) recursive function. The part of mechanising the translation of recursive function to register machines has already been done by Zammit in HOL4 [10], although his register machines use a slightly different instruction set than the one described in [2].

Contributions: We formalised in Isabelle/HOL Turing machines following the description of Boolos et al [2] where tapes only have blank or occupied cells. We mechanise the undecidability of the halting problem and prove the correctness of concrete Turing machines that are needed in this proof; such correctness proofs are left out in the informal literature. For reasoning about Turing machine programs we derive Hoare-rules. We also construct the universal Turing machine from [2] by translating recursive

functions to abacus machines and abacus machines to Turing machines. This works essentially like a small, verified compiler from recursive functions to Turing machine programs. When formalising the universal Turing machine, we stumbled in [2] upon an inconsistent use of the definition of what partial function a Turing machine calculates.

2 Turing Machines

Turing machines can be thought of as having a *head*, "gliding" over a potentially infinite tape. Boolos et al [2] only consider tapes with cells being either blank or occupied, which we represent by a datatype having two constructors, namely Bk and Oc. One way to represent such tapes is to use a pair of lists, written (l, r), where l stands for the tape on the left-hand side of the head and r for the tape on the right-hand side. We use the notation Bk^n (similarly Oc^n) for lists composed of n elements of Bks. We also have the convention that the head, abbreviated *hd*, of the right list is the cell on which the head of the Turing machine currently scans. This can be pictured as follows:



Note that by using lists each side of the tape is only finite. The potential infinity is achieved by adding an appropriate blank or occupied cell whenever the head goes over the "edge" of the tape. To make this formal we define five possible *actions* the Turing machine can perform:

$$\begin{array}{rcll} a & ::= & W_{Bk} & \text{(write blank, } Bk\text{)} \\ & | & W_{Oc} & \text{(write occupied, } Oc\text{)} \\ & | & L & \text{(move left)} \\ & | & R & \text{(move right)} \\ & | & Nop & \text{(do-nothing operation)} \end{array}$$

We slightly deviate from the presentation in [2] (and also [1]) by using the *Nop* operation; however its use will become important when we formalise halting computations and also universal Turing machines. Given a tape and an action, we can define the following tape updating function:

$$\begin{array}{lcl} update\ (l, r)\ W_{Bk} & \stackrel{def}{=} & (l, Bk::tl\ r) \\ update\ (l, r)\ W_{Oc} & \stackrel{def}{=} & (l, Oc::tl\ r) \\ update\ (l, r)\ L & \stackrel{def}{=} & if\ l = []\ then\ ([], Bk::r)\ else\ (tl\ l, hd\ l::r) \\ update\ (l, r)\ R & \stackrel{def}{=} & if\ r = []\ then\ (Bk::l, [])\ else\ (hd\ r::l, tl\ r) \\ update\ (l, r)\ Nop & \stackrel{def}{=} & (l, r) \end{array}$$

The first two clauses replace the head of the right list with a new Bk or Oc, respectively. To see that these two clauses make sense in case where r is the empty list, one has to

know that the tail function, *tl*, is defined such that $tl\ [] \stackrel{def}{=} []$ holds. The third clause implements the move of the head one step to the left: we need to test if the left-list l is empty; if yes, then we just prepend a blank cell to the right list; otherwise we have to remove the head from the left-list and prepend it to the right list. Similarly in the fourth clause for a right move action. The *Nop* operation leaves the tape unchanged.

Next we need to define the *states* of a Turing machine. We follow the choice made in [1] by representing a state with a natural number and the states in a Turing machine program by the initial segment of natural numbers starting from *0*. In doing so we can compose two Turing machine programs by shifting the states of one by an appropriate amount to a higher segment and adjusting some "next states" in the other.

An *instruction* of a Turing machine is a pair consisting of an action and a natural number (the next state). A *program p* of a Turing machine is then a list of such pairs. Using as an example the following Turing machine program, which consists of four instructions

$$dither \stackrel{def}{=} [\overbrace{\underbrace{(W_{Bk}, 1)}_{Bk\text{-case}}, \underbrace{(R, 2)}_{Oc\text{-case}}}^{\text{1st state = starting state}}, \underbrace{(L, 1), (L, 0)}_{\text{2nd state}}] \tag{1}$$

the reader can see we have organised our Turing machine programs so that segments of two pairs belong to a state. The first component of such a segment determines what action should be taken and which next state should be transitioned to in case the head reads a *Bk*; similarly the second component determines what should be done in case of reading *Oc*. We have the convention that the first state is always the *starting state* of the Turing machine. The *0*-state is special in that it will be used as the "halting state". There are no instructions for the *0*-state, but it will always perform a *Nop*-operation and remain in the *0*-state. We have chosen a very concrete representation for Turing machine programs, because when constructing a universal Turing machine, we need to define a coding function for programs.

Given a program *p*, a state and the cell being scanned by the head, we need to fetch the corresponding instruction from the program. For this we define the function *fetch*

$$\begin{array}{l} fetch\ p\ 0\ \_ = (Nop, 0) \\ fetch\ p\ (Suc\ s)\ Bk \stackrel{def}{=} case\ nth\_of\ p\ (2 * s)\ of \\ \qquad\qquad\qquad\qquad None \Rightarrow (Nop, 0) \mid Some\ i \Rightarrow i \\ fetch\ p\ (Suc\ s)\ Oc \stackrel{def}{=} case\ nth\_of\ p\ (2 * s + 1)\ of \\ \qquad\qquad\qquad\qquad None \Rightarrow (Nop, 0) \mid Some\ i \Rightarrow i \end{array} \tag{2}$$

In this definition the function *nth_of* returns the *n*th element from a list, provided it exists (*Some*-case), or if it does not, it returns the default action *Nop* and the default state *0* (*None*-case). We often have to restrict Turing machine programs to be well-formed: a program *p* is *well-formed* if it satisfies the following three properties:

$$wf\ p \stackrel{def}{=} 2 \leq length\ p \wedge is\_even\ (length\ p) \wedge (\forall\, (a, s) \in p.\ s \leq length\ p\ div\ 2)$$

The first states that p must have at least an instruction for the starting state; the second that p has a Bk and Oc instruction for every state, and the third that every next-state is one of the states mentioned in the program or being the 0-state.

A *configuration* c of a Turing machine is a state together with a tape. This is written as $(s, (l, r))$. We say a configuration *is final* if $s = 0$ and we say a predicate P *holds for* a configuration if P holds for the tape (l, r). If we have a configuration and a program, we can calculate what the next configuration is by fetching the appropriate action and next state from the program, and by updating the state and tape accordingly. This single step of execution is defined as the function *step*

$$\begin{array}{l} step\ (s, (l, r))\ p \stackrel{def}{=} let\ (a, s') = fetch\ p\ s\ (read\ r) \\ \qquad\qquad\qquad\qquad\quad in\ (s', update\ (l, r)\ a) \end{array}$$

where *read* r returns the head of the list r, or if r is empty it returns Bk. We lift this definition to an evaluation function that performs exactly n steps:

$$\begin{array}{lcl} steps\ c\ p\ 0 & \stackrel{def}{=} & c \\ steps\ c\ p\ (Suc\ n) & \stackrel{def}{=} & steps\ (step\ c\ p)\ p\ n \end{array}$$

Recall our definition of *fetch* (shown in (2)) with the default value for the 0-state. In case a Turing program takes according to the usual textbook definition, say [2], less than n steps before it halts, then in our setting the *steps*-evaluation does not actually halt, but rather transitions to the 0-state (the final state) and remains there performing *Nop*-actions until n is reached.

We often need to restrict tapes to be in standard form, which means the left list of the tape is either empty or only contains Bks, and the right list contains some "clusters" of Ocs separated by single blanks. To make this formal we define the following overloaded function encoding natural numbers into lists of Ocs and Bks.

$$\begin{array}{lcl@{\qquad}lcl} \langle n\rangle & \stackrel{def}{=} & Oc^{n+1} & \langle []\rangle & \stackrel{def}{=} & [] \\ \langle (n, m)\rangle & \stackrel{def}{=} & \langle n\rangle\ @\ [Bk]\ @\ \langle m\rangle & \langle [n]\rangle & \stackrel{def}{=} & \langle n\rangle \\ & & & \langle n::ns\rangle & \stackrel{def}{=} & \langle (n, ns)\rangle \end{array} \tag{3}$$

A *standard tape* is then of the form $(Bk^k, \langle [n_1,...,n_m]\rangle\ @\ Bk^l)$ for some k, l and $n_{1...m}$. Note that the head in a standard tape "points" to the leftmost Oc on the tape. Note also that the natural number 0 is represented by a single filled cell on a standard tape, 1 by two filled cells and so on.

We need to be able to sequentially compose Turing machine programs. Given our concrete representation, this is relatively straightforward, if slightly fiddly. We use the following two auxiliary functions:

$$\begin{array}{l} shift\ p\ n \stackrel{def}{=} map\ (\lambda(a, s).\ (a, if\ s = 0\ then\ 0\ else\ s + n))\ p \\ adjust\ p \stackrel{def}{=} map\ (\lambda(a, s).\ (a, if\ s = 0\ then\ Suc\ (length\ p\ div\ 2)\ else\ s))\ p \end{array}$$

The first adds n to all states, except the 0-state, thus moving all "regular" states to the segment starting at n; the second adds $Suc\ (length\ p\ div\ 2)$ to the 0-state, thus redirecting

$$cbegin \stackrel{def}{=} [(W_{Bk}, 0), (R, 2), (R, 3), (R, 2), (W_{Oc}, 3), (L, 4), (L, 4), (L, 0)]$$

$$cloop \stackrel{def}{=} [(R, 0), (R, 2), (R, 3), (W_{Bk}, 2), (R, 3), (R, 4), (W_{Oc}, 5), (R, 4), (L, 6), (L, 5), (L, 6), (L, 1)]$$

$$cend \stackrel{def}{=} [(L, 0), (R, 2), (W_{Oc}, 3), (L, 4), (R, 2), (R, 2), (L, 5), (W_{Bk}, 4), (R, 0), (L, 5)]$$

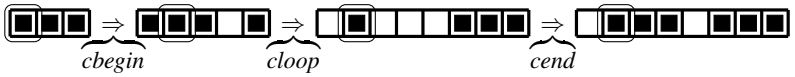


Fig. 1. The three components of the *copy Turing machine* (above). If started (below) with the tape $([], \langle 2 \rangle)$ the first machine appends $[Bk, Oc]$ at the end of the right tape; the second then "moves" all Ocs except the first from the beginning of the tape to the end; the third "refills" the original block of Ocs. The resulting tape is $([Bk], \langle (2, 2) \rangle)$.

all references to the "halting state" to the first state after the program p. With these two functions in place, we can define the *sequential composition* of two Turing machine programs p_1 and p_2 as

$$p_1 ; p_2 \stackrel{def}{=} adjust\ p_1\ @\ shift\ p_2\ (length\ p_1\ div\ 2)$$

Before we can prove the undecidability of the halting problem for our Turing machines working on standard tapes, we have to analyse two concrete Turing machine programs and establish that they are correct—that means they are "doing what they are supposed to be doing". Such correctness proofs are usually left out in the informal literature, for example [2]. The first program we need to prove correct is the *dither* program shown in (1) and the second program is *copy* defined as

$$copy \stackrel{def}{=} cbegin ; cloop ; cend \tag{4}$$

whose three components are given in Figure 1. For our correctness proofs, we introduce the notion of total correctness defined in terms of *Hoare-triples*, written $\{P\}\ p\ \{Q\}$. They implement the idea that a program p started in state *1* with a tape satisfying P will after some n steps halt (have transitioned into the halting state) with a tape satisfying Q. This idea is very similar to the notion of *realisability* in [1]. We also have *Hoare-pairs* of the form $\{P\}\ p \uparrow$ implementing the case that a program p started with a tape satisfying P will loop (never transition into the halting state). Both notion are formally defined as

$$\{P\}\ p\ \{Q\} \stackrel{def}{=} \forall tp.\ \text{if } P\ tp \text{ holds then } \exists n.\ \text{such that } is\_final\ (steps\ (1, tp)\ p\ n)\ \wedge\ Q\ holds\_for\ (steps\ (1, tp)\ p\ n)$$

$$\{P\}\ p \uparrow \stackrel{def}{=} \forall tp.\ \text{if } P\ tp \text{ holds then } \forall n.\ \neg\ is\_final\ (steps\ (1, tp)\ p\ n)$$

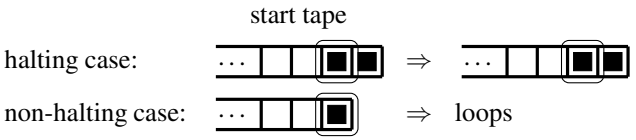
For our Hoare-triples we can easily prove the following Hoare-consequence rule

$$\frac{P' \mapsto P \quad \{P\}\, p\, \{Q\} \quad Q \mapsto Q'}{\{P'\}\, p\, \{Q'\}} \tag{5}$$

where $P' \mapsto P$ stands for the fact that for all tapes tp, $P'\, tp$ implies $P\, tp$ (similarly for Q and Q').

Like Asperti and Ricciotti with their notion of realisability, we have set up our Hoare-rules so that we can deal explicitly with total correctness and non-termination, rather than have notions for partial correctness and termination. Although the latter would allow us to reason more uniformly (only using Hoare-triples), we prefer our definitions because we can derive below some simple Hoare-rules for sequentially composed Turing programs. In this way we can reason about the correctness of *cbegin*, for example, completely separately from *cloop* and *cend*.

It is relatively straightforward to prove that the Turing program *dither* shown in (1) is correct. This program should be the "identity" when started with a standard tape representing *1* but loops when started with the *0*-representation instead, as pictured below.



We can prove the following two Hoare-statements:

$$\{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1 \rangle)\}\ dither\ \{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1 \rangle)\}$$
$$\{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 0 \rangle)\}\ dither \uparrow$$

The first is by a simple calculation. The second is by an induction on the number of steps we can perform starting from the input tape.

The program *copy* defined in (4) has 15 states; its purpose is to produce the standard tape $(Bks, \langle (n, n) \rangle)$ when started with $(Bks, \langle n \rangle)$, that is making a copy of a value n on the tape. Reasoning about this program is substantially harder than about *dither*. To ease the burden, we derive the following two Hoare-rules for sequentially composed programs.

$$\frac{\{P\}\, p_1\, \{Q\} \quad \{Q\}\, p_2\, \{R\}}{\{P\}\, p_1\, ;p_2\, \{R\}}\ wf\, p_1 \qquad \frac{\{P\}\, p_1\, \{Q\} \quad \{Q\}\, p_2 \uparrow}{\{P\}\, p_1\, ;p_2 \uparrow}\ wf\, p_1$$

The first corresponds to the usual Hoare-rule for composition of two terminating programs. The second rule gives the conditions for when the first program terminates generating a tape for which the second program loops. The side-conditions about $wf\, p_1$ are needed in order to ensure that the redirection of the halting and initial state in p_1 and p_2, respectively, match up correctly. These Hoare-rules allow us to prove the correctness of *copy* by considering the correctness of the components *cbegin*, *cloop* and *cend* in isolation. This simplifies the reasoning considerably, for example when designing

$$
\begin{array}{lll}
I_1\ n\ (l, r) & \stackrel{def}{=} & (l, r) = ([], Oc^n) \quad \text{(starting state)} \\
I_2\ n\ (l, r) & \stackrel{def}{=} & \exists i\, j.\ 0 < i \wedge i + j = n \wedge (l, r) = (Oc^i, Oc^j) \\
I_3\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, tl\ r) = (Bk::Oc^n, []) \\
I_4\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = (Oc^n, [Bk, Oc]) \vee (l, r) = (Oc^{n-1}, [Oc, Bk, Oc]) \\
I_0\ n\ (l, r) & \stackrel{def}{=} & 1 < n \wedge (l, r) = (Oc^{n-2}, [Oc, Oc, Bk, Oc]) \vee \quad \text{(halting state)} \\
 & & n = 1 \wedge (l, r) = ([], [Bk, Oc, Bk, Oc]) \\
\hline
J_1\ n\ (l, r) & \stackrel{def}{=} & \exists i\, j.\ i + j + 1 = n \wedge (l, r) = (Oc^i, Oc::Oc::Bk^j\ @\ Oc^j) \wedge 0 < j \vee \\
 & & 0 < n \wedge (l, r) = ([], Bk::Oc::Bk^n\ @\ Oc^n) \quad \text{(starting state)} \\
J_0\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = ([Bk], Oc::Bk^n\ @\ Oc^n) \quad \text{(halting state)} \\
\hline
K_1\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = ([Bk], Oc::Bk^n\ @\ Oc^n) \quad \text{(starting state)} \\
K_0\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = ([Bk], Oc^n\ @\ (Bk::Oc^n)) \quad \text{(halting state)}
\end{array}
$$

Fig. 2. The invariants I_0,...,I_4 are for the states of *cbegin*. Below, the invariants only for the starting and halting states of *cloop* and *cend* are shown. In each invariant, the parameter *n* stands for the number of *Oc*s with which the Turing machine is started.

decreasing measures for proving the termination of the programs. We will show the details for the program *cbegin*. For the two other programs we refer the reader to our formalisation.

Given the invariants I_0,..., I_4 shown in Figure 2, which correspond to each state of *cbegin*, we define the following invariant for the whole *cbegin* program:

$$
\begin{array}{lll}
I_{cbegin}\ n\ (s, tp) & \stackrel{def}{=} & \textit{if } s = 0 \textit{ then } I_0\ n\ tp \\
 & & \textit{else if } s = 1 \textit{ then } I_1\ n\ tp \\
 & & \textit{else if } s = 2 \textit{ then } I_2\ n\ tp \\
 & & \textit{else if } s = 3 \textit{ then } I_3\ n\ tp \\
 & & \textit{else if } s = 4 \textit{ then } I_4\ n\ tp \\
 & & \textit{else False}
\end{array}
$$

This invariant depends on *n* representing the number of *Oc*s on the tape. It is not hard (26 lines of automated proof script) to show that for $0 < n$ this invariant is preserved under the computation rules *step* and *steps*. This gives us partial correctness for *cbegin*.

We next need to show that *cbegin* terminates. For this we introduce lexicographically ordered pairs (n, m) derived from configurations $(s, (l, r))$ whereby *n* is the state *s*, but ordered according to how *cbegin* executes them, that is $1 > 2 > 3 > 4 > 0$; in order to have a strictly decreasing measure, *m* takes the data on the tape into account and is calculated according to the following measure function:

$$
\begin{array}{lll}
M_{cbegin}(s, (l, r)) & \stackrel{def}{=} & \textit{if } s = 2 \textit{ then length } r \\
 & & \textit{else if } s = 3 \textit{ then } (\textit{if } r = [] \vee r = [Bk] \textit{ then 1 else 0}) \\
 & & \textit{else if } s = 4 \textit{ then length } l \\
 & & \textit{else 0}
\end{array}
$$

With this in place, we can show that for every starting tape of the form $([], Oc^n)$ with $0 < n$, the Turing machine *cbegin* will eventually halt (the measure decreases in each step). Taking this and the partial correctness proof together, we obtain the Hoare-triple shown on the left for *cbegin*:

$$\{I_1\ n\}\ cbegin\ \{I_0\ n\} \qquad \{J_1\ n\}\ cloop\ \{J_0\ n\} \qquad \{K_1\ n\}\ cend\ \{K_0\ n\}$$

where we assume $0 < n$ (similar reasoning is needed for the Hoare-triples for *cloop* and *cend*). Since the invariant of the halting state of *cbegin* implies the invariant of the starting state of *cloop*, that is $I_0\ n \mapsto J_1\ n$ holds, and also $J_0\ n = K_1\ n$, we can derive the following Hoare-triple for the correctness of *copy*:

$$\{\lambda tp.\ tp = ([], \langle n\rangle)\}\ copy\ \{\lambda tp.\ tp = ([Bk], \langle(n, n)\rangle)\}$$

That means if we start with a tape of the form $([], \langle n\rangle)$ then *copy* will halt with the tape $([Bk], \langle(n, n)\rangle)$, as desired.

Finally, we are in the position to prove the undecidability of the halting problem. A program p started with a standard tape containing the (encoded) numbers *ns* will *halt* with a standard tape containing a single (encoded) number is defined as

$$halts\ p\ ns \stackrel{def}{=} \{\lambda tp.\ tp = ([], \langle ns\rangle)\}\ p\ \{\lambda tp.\ \exists k\ n\ l.\ tp = (Bk^k, \langle n\rangle\ @\ Bk^l)\}$$

This roughly means we considering only Turing machine programs representing functions that take some numbers as input and produce a single number as output. For undecidability, the property we are proving is that there is no Turing machine that can decide in general whether a Turing machine program halts (answer either *0* for halting or *1* for looping). Given our correctness proofs for *dither* and *copy* shown above, this non-existence is now relatively straightforward to establish. We first assume there is a coding function, written *code M*, which represents a Turing machine M as a natural number. No further assumptions are made about this coding function. Suppose a Turing machine H exists such that if started with the standard tape $([Bk], \langle(code\ M, ns)\rangle)$ returns *0*, respectively *1*, depending on whether M halts or not when started with the input tape containing $\langle ns\rangle$. This assumption is formalised as follows—for all M and all lists of natural numbers *ns*:

$$\begin{array}{l} halts\ M\ ns \text{ implies } \{\lambda tp.\ tp = ([Bk], \langle(code\ M, ns)\rangle)\}\ H\ \{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 0\rangle)\} \\ \neg\ halts\ M\ ns \text{ implies } \{\lambda tp.\ tp = ([Bk], \langle(code\ M, ns)\rangle)\}\ H\ \{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1\rangle)\} \end{array}$$

The contradiction can be derived using the following Turing machine

$$contra \stackrel{def}{=} copy\ ;\ H\ ;\ dither$$

Suppose *halts contra* [*code contra*] holds. Given the invariants $P_1 \ldots P_3$ shown on the left, we can derive the following Hoare-pair for *contra* on the right.

$$\begin{array}{l} P_1 \stackrel{def}{=} \lambda tp.\ tp = ([], \langle code\ contra\rangle) \\ P_2 \stackrel{def}{=} \lambda tp.\ tp = ([Bk], \langle(code\ contra, code\ contra)\rangle) \\ P_3 \stackrel{def}{=} \lambda tp.\ \exists k.\ tp = (Bk^k, \langle 0\rangle) \end{array} \qquad \dfrac{\dfrac{\{P_1\}\ copy\ \{P_2\} \quad \{P_2\}\ H\ \{P_3\}}{\{P_1\}\ copy\ ;\ H\ \{P_3\}} \qquad \{P_3\}\ dither\uparrow}{\{P_1\}\ contra\uparrow}$$

This Hoare-pair contradicts our assumption that *contra* started with $\langle code\ contra\rangle$ halts.

Suppose $\neg$ *halts contra* [*code contra*] holds. Again, given the invariants $Q_1 \ldots Q_3$ shown on the left, we can derive the Hoare-triple for *contra* on the right.

$$Q_1 \stackrel{def}{=} \lambda tp.\ tp = ([], \langle code\ contra\rangle)$$
$$Q_2 \stackrel{def}{=} \lambda tp.\ tp = ([Bk], \langle (code\ contra, code\ contra)\rangle)$$
$$Q_3 \stackrel{def}{=} \lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1\rangle)$$

$$\dfrac{\dfrac{\{Q_1\}\ copy\ \{Q_2\} \quad \{Q_2\}\ H\ \{Q_3\}}{\{Q_1\}\ copy\ ;\ H\ \{Q_3\}} \qquad \{Q_3\}\ dither\ \{Q_3\}}{\{Q_1\}\ contra\ \{Q_3\}}$$

This time the Hoare-triple states that *contra* terminates with the "output" $\langle 1\rangle$. In both cases we come to a contradiction, which means we have to abandon our assumption that there exists a Turing machine H which can in general decide whether Turing machines terminate.

3 Abacus Machines

Boolos et al [2] use abacus machines as a stepping stone for making it less laborious to write Turing machine programs. Abacus machines operate over a set of registers R_0, R_1, ..., R_n each being able to hold an arbitrary large natural number. We use natural numbers to refer to registers; we also use a natural number to represent a program counter and to represent jumping "addresses", for which we use the letter l. An abacus program is a list of *instructions* defined by the datatype:

$i ::=$	*Inc R*	increment register R by one
|	*Dec R l*	if content of R is non-zero, then decrement it by one otherwise jump to instruction l
|	*Goto l*	jump to instruction l

For example the program clearing the register R (that is setting it to *0*) can be defined as follows:

$$clear\ R\ l \stackrel{def}{=} [Dec\ R\ l, Goto\ 0]$$

Running such a program means we start with the first instruction then execute one instructions after the other, unless there is a jump. For example the second instruction *Goto 0* above means we jump back to the first instruction thereby closing the loop. Like with our Turing machines, we fetch instructions from an abacus program such that a jump out of "range" behaves like a *Nop*-action. In this way it is again easy to define a function *steps* that executes n instructions of an abacus program. A *configuration* of an abacus machine is the current program counter together with a snapshot of all registers. By convention the value calculated by an abacus program is stored in the last register (the one with the highest index in the program).

The main point of abacus programs is to be able to translate them to Turing machine programs. Registers and their content are represented by standard tapes (see definition

shown in (3)). Because of the jumps in abacus programs, it is impossible to build Turing machine programs out of components using our ;-operation shown in the previous section. To overcome this difficulty, we calculate a *layout* of an abacus program as follows

$$\begin{array}{lcl} \mathit{layout}\ [] & \stackrel{\mathit{def}}{=} & [] \\ \mathit{layout}\ (\mathit{Inc}\ R::\mathit{is}) & \stackrel{\mathit{def}}{=} & 2 * R + 9::\mathit{layout}\ \mathit{is} \\ \mathit{layout}\ (\mathit{Dec}\ R\ l::\mathit{is}) & \stackrel{\mathit{def}}{=} & 2 * R + 16::\mathit{layout}\ \mathit{is} \\ \mathit{layout}\ (\mathit{Goto}\ l::\mathit{is}) & \stackrel{\mathit{def}}{=} & 1::\mathit{layout}\ \mathit{is} \end{array}$$

This gives us a list of natural numbers specifying how many states are needed to translate each abacus instruction. This information is needed in order to calculate the state where the Turing machine program of an abacus instruction starts. This can be defined as

$$\mathit{address}\ p\ n = \mathit{Suc}\ (\Sigma\ (\mathit{take}\ n\ (\mathit{layout}\ p)))$$

where p is an abacus program and *take n* takes the first n elements from a list.

The *Goto* instruction is easiest to translate requiring only one state, namely the Turing machine program:

$$\mathit{translate\_Goto}\ l \stackrel{\mathit{def}}{=} [(\mathit{Nop}, l), (\mathit{Nop}, l)]$$

where l is the state in the Turing machine program to jump to. For translating the instruction *Inc R*, one has to remember that the content of the registers are encoded in the Turing machine as a standard tape. Therefore the translated Turing machine needs to first find the number corresponding to the content of register R. This needs a machine with $2 * R$ states and can be constructed as follows:

$$\begin{array}{l} \mathit{TMFindnth}\ 0 \stackrel{\mathit{def}}{=} [] \\ \mathit{TMFindnth}\ (\mathit{Suc}\ n) \stackrel{\mathit{def}}{=} \\ \quad \mathit{TMFindnth}\ n\ @\ [(W_{Oc}, 2 * n + 1), (R, 2 * n + 2), (R, 2 * n + 3), (R, 2 * n + 2)] \end{array}$$

Then we need to increase the "number" on the tape by one, and adjust the following "registers". For adjusting we only need to change the first *Oc* of each number to *Bk* and the last one from *Bk* to *Oc*. Finally, we need to transition the head of the Turing machine back into the standard position. This requires a Turing machine with 9 states (we omit the details). Similarly for the translation of *Dec R l*, where the translated Turing machine needs to first check whether the content of the corresponding register is *0*. For this we have a Turing machine program with *16* states (again the details are omitted).

Finally, having a Turing machine for each abacus instruction we need to "stitch" the Turing machines together into one so that each Turing machine component transitions to next one, just like in the abacus programs. One last problem to overcome is that an abacus program is assumed to calculate a value stored in the last register (the one with the highest register). That means we have to append a Turing machine that "mops up" the tape (cleaning all *Oc*s) except for the *Oc*s of the last register represented on the tape.

This needs a Turing machine program with $2 * R + 6$ states, assuming R is the number of registers to be "cleaned".

While generating the Turing machine program for an abacus program is not too difficult to formalise, the problem is that it contains *Goto*s all over the place. The unfortunate result is that we cannot use our Hoare-rules for reasoning about sequentially composed programs (for this each component needs to be completely independent). Instead we have to treat the translated Turing machine as one "big block" and prove as invariant that it performs the same operations as the abacus program. For this we have to show that for each configuration of an abacus machine the *step*-function is simulated by zero or more steps in our translated Turing machine. This leads to a rather large "monolithic" correctness proof (4600 loc and 380 sublemmas) that on the conceptual level is difficult to break down into smaller components.

4 Recursive Functions and a Universal Turing Machine

The main point of recursive functions is that we can relatively easily construct a universal Turing machine via a universal function. This is different from Norrish [7] who gives a universal function for the lambda-calculus, and also from Asperti and Ricciotti [1] who construct a universal Turing machine directly, but for simulating Turing machines with a more restricted alphabet. Unlike Norrish [7], we need to represent recursive functions "deeply" because we want to translate them to abacus programs. Thus *recursive functions* are defined as the datatype

$$\begin{array}{lll} r ::= z & \text{(zero-function)} & \mid Cn^n\, f\, fs \quad \text{(composition)} \\ \mid s & \text{(successor-function)} & \mid Pr^n\, f_1\, f_2 \quad \text{(primitive recursion)} \\ \mid id^n_m & \text{(projection)} & \mid Mn^n\, f \quad \text{(minimisation)} \end{array}$$

where n indicates the function expects n arguments (in [2] both z and s expect one argument), and *fs* stands for a list of recursive functions. Since we know in each case the arity, say l, we can define an evaluation function, called *eval*, that takes a recursive function f and a list *ns* of natural numbers of length l as arguments. Since this evaluation function uses the minimisation operator from HOL, this function might not terminate always. As a result we also need to inductively characterise when *eval* terminates. We omit the definitions for *eval f ns* and *terminate f ns*. Because of space reasons, we also omit the definition of translating recursive functions into abacus programs. We can prove, however, the following theorem about the translation: If *terminate f ns* holds for the recursive function f and arguments *ns*, then the following Hoare-triple holds

$$\{\lambda tp.\ tp = ([Bk, Bk], \langle ns\rangle)\}\ translate\ f\ \{\lambda tp.\ \exists\, k\, l.\ tp = (Bk^k, \langle eval\, f\, ns\rangle\ @\ Bk^l)\}$$

for the Turing machine generated by *translate f*. This means the translated Turing machine if started with the standard tape $([Bk, Bk], \langle ns\rangle)$ will terminate with the standard tape $(Bk^k, \langle eval\, f\, ns\rangle\ @\ Bk^l)$ for some k and l.

Having recursive functions under our belt, we can construct a universal function, written *UF*. This universal function acts like an interpreter for Turing machines. It takes two arguments: one is the code of the Turing machine to be interpreted and the other

is the "packed version" of the arguments of the Turing machine. We can then consider how this universal function is translated to a Turing machine and from this construct the universal Turing machine, written *UTM*. It is defined as the composition of the Turing machine that packages the arguments and the translated recursive function *UF*:

$$UTM \stackrel{def}{=} arg\_coding\ ;\ (translate\ UF)$$

Suppose a Turing program p is well-formed and when started with the standard tape containing the arguments *args*, will produce a standard tape with "output" n. This assumption can be written as the Hoare-triple

$$\{\lambda tp.\ tp = ([], \langle args\rangle)\}\ p\ \{\lambda tp.\ tp = (Bk^m, \langle n\rangle\ @\ Bk^k)\}$$

where we require that the *args* stand for a non-empty list. Then the universal Turing machine *UTM* started with the code of p and the arguments *args*, calculates the same result, namely

$$\{\lambda tp.\ tp = ([], \langle code\ p::args\rangle)\}\ UTM\ \{\lambda tp.\ \exists m\ k.\ tp = (Bk^m, \langle n\rangle\ @\ Bk^k)\}$$

Similarly, if a Turing program p started with the standard tape containing *args* loops, which is represented by the Hoare-pair

$$\{\lambda tp.\ tp = ([], \langle args\rangle)\}\ p\uparrow$$

then the universal Turing machine started with the code of p and the arguments *args* will also loop

$$\{\lambda tp.\ tp = ([], \langle code\ p::args\rangle)\}\ UTM\uparrow$$

While formalising the chapter in [2] about universal Turing machines, an unexpected outcome of our work is that we identified an inconsistency in their use of a definition. This is unexpected since [2] is a classic textbook which has undergone several editions (we used the fifth edition; the material containing the inconsistency was introduced in the fourth edition of this book). The central idea about Turing machines is that when started with standard tapes they compute a partial arithmetic function. The inconsistency arises when they define the case when this function should *not* return a result. Boolos at al write in Chapter 3, Page 32:

> *"If the function that is to be computed assigns no value to the arguments that are represented initially on the tape, then the machine either will never halt, or will halt in some nonstandard configuration..."*

Interestingly, they do not implement this definition when constructing their universal Turing machine. In Chapter 8, on page 93, a recursive function *stdh* is defined as:

$$stdh(m, x, t) \stackrel{def}{=} stat(conf(m, x, t)) + nstd(conf(m, x, t)) \tag{6}$$

where $stat(conf(m, x, t))$ computes the current state of the simulated Turing machine, and $nstd(conf(m, x, t))$ returns *1* if the tape content is non-standard. If either one evaluates to something that is not zero, then $stdh(m, x, t)$ will be not zero, because of the

+-operation. On the same page, a function $halt(m, x)$ is defined in terms of *stdh* for computing the steps the Turing machine needs to execute before it halts (in case it halts at all). According to this definition, the simulated Turing machine will continue to run after entering the *0*-state with a non-standard tape. The consequence of this inconsistency is that there exist Turing machines that given some arguments do not compute a value according to Chapter 3, but return a proper result according to the definition in Chapter 8. One such Turing machine is:

$$counter\_example \stackrel{def}{=} [(L, 0), (L, 2), (R, 2), (R, 0)]$$

If started with standard tape $([], [Oc])$, it halts with the non-standard tape $([Oc, Bk], [])$ according to the definition in Chapter 3—so no result is calculated; but with the standard tape $([Bk], [Oc])$ according to the definition in Chapter 8. We solve this inconsistency in our formalisation by setting up our definitions so that the *counter_example* Turing machine does not produce any result by looping forever fetching *Nop*s in state *0*. This solution implements essentially the definition in Chapter 3; it differs from the definition in Chapter 8, where perplexingly the instruction from state *1* is fetched.

5 Conclusion

In previous works we were unable to formalise results about computability because in Isabelle/HOL we cannot, for example, represent the decidability of a predicate P, say, as the formula $P \vee \neg P$. For reasoning about computability we need to formalise a concrete model of computations. We could have followed Norrish [7] using the λ-calculus as the starting point for formalising computability theory, but then we would have to reimplement on the ML-level his infrastructure for rewriting λ-terms modulo β-equality: HOL4 has a simplifer that can rewrite terms modulo an arbitrary equivalence relation, which Isabelle unfortunately does not yet have. Even though, we would still need to connect λ-terms somehow to Turing machines for proofs that make essential use of them (for example the undecidability proof for Wang's tiling problem [9]).

We therefore have formalised Turing machines in the first place and the main computability results from Chapters 3 to 8 in the textbook by Boolos et al [2]. For this we did not need to implement anything on the ML-level of Isabelle/HOL. While formalising the six chapters of [2] we have found an inconsistency in Boolos et al's definitions of what function a Turing machine calculates. In Chapter 3 they use a definition that states a function is undefined if the Turing machine loops *or* halts with a non-standard tape. Whereas in Chapter 8 about the universal Turing machine, the Turing machines will *not* halt unless the tape is in standard form. Like Nipkow [6] observed with his formalisation of a textbook, we found that Boolos et al are (almost) right. We have not attempted to formalise everything precisely as Boolos et al present it, but use definitions that make our mechanised proofs manageable. For example our definition of the halting state performing *Nop*-operations seems to be non-standard, but very much suited to a formalisation in a theorem prover where the *steps*-function needs to be total.

Norrish mentions that formalising Turing machines would be a "*daunting prospect*" [7, Page 310]. While λ-terms indeed lead to some slick mechanised proofs, our experience is that Turing machines are not too daunting if one is only concerned with

formalising the undecidability of the halting problem for Turing machines. As a point of comparison, the halting problem took us around 1500 loc of Isar-proofs, which is just one-and-a-half times of a mechanised proof pearl about the Myhill-Nerode theorem. So our conclusion is that this part is not as daunting as we estimated when reading the paper by Norrish [7]. The work involved with constructing a universal Turing machine via recursive functions and abacus machines, we agree, is not a project one wants to undertake too many times (our formalisation of abacus machines and their correct translation is approximately 4600 loc; recursive functions 2800 loc and the universal Turing machine 10000 loc).

Our work is also very much inspired by the formalisation of Turing machines of Asperti and Ricciotti [1] in the Matita theorem prover. It turns out that their notion of realisability and our Hoare-triples are very similar, however we differ in some basic definitions for Turing machines. Asperti and Ricciotti are interested in providing a mechanised foundation for complexity theory. They formalised a universal Turing machine (which differs from ours by using a more general alphabet), but did not describe an undecidability proof. Given their definitions and infrastructure, we expect however this should not be too difficult for them.

For us the most interesting aspects of our work are the correctness proofs for Turing machines. Informal presentations of computability theory often leave the constructions of particular Turing machines as exercise to the reader, for example [2], deeming it to be just a chore. However, as far as we are aware all informal presentations leave out any arguments why these Turing machines should be correct. This means the reader is left with the task of finding appropriate invariants and measures for showing the correctness and termination of these Turing machines. Whenever we can use Hoare-style reasoning, the invariants are relatively straightforward and again as a point of comparison much smaller than for example the invariants used by Myreen in a correctness proof of a garbage collector written in machine code [5, Page 76]. However, the invariant needed for the abacus proof, where Hoare-style reasoning does not work, is similar in size as the one by Myreen and finding a sufficiently strong one took us, like Myreen, something on the magnitude of weeks.

Our reasoning about the invariants is not much supported by the automation beyond the standard automation tools available in Isabelle/HOL. There is however a tantalising connection between our work and very recent work by Jensen et al [4] on verifying X86 assembly code that might change that. They observed a similar phenomenon with assembly programs where Hoare-style reasoning is sometimes possible, but sometimes it is not. In order to ease their reasoning, they introduced a more primitive specification logic, on which Hoare-rules can be provided for special cases. It remains to be seen whether their specification logic for assembly code can make it easier to reason about our Turing programs. That would be an attractive result, because Turing machine programs are very much like assembly programs and it would connect some very classic work on Turing machines to very cutting-edge work on machine code verification. In order to try out such ideas, our formalisation provides the "playground". The code of our formalisation is available from the Mercurial repository at http://www.dcs.kcl.ac.uk/staff/urbanc/cgi-bin/repos.cgi/tm/.

Acknowledgements. We are very grateful for the extremely helpful comments by the anonymous reviewers.

References

1. Asperti, A., Ricciotti, W.: Formalizing Turing Machines. In: Ong, L., de Queiroz, R. (eds.) WoLLIC 2012. LNCS, vol. 7456, pp. 1–25. Springer, Heidelberg (2012)
2. Boolos, G., Burgess, J.P., Jeffrey, R.C.: Computability and Logic, 5th edn. Cambridge University Press (2007)
3. Dijkstra, E.W.: Go to Statement Considered Harmful. Communications of the ACM 11(3), 147–148 (1968)
4. Jensen, J.B., Benton, N., Kennedy, A.: High-Level Separation Logic for Low-Level Code. In: Proc. of the 40th Symposium on Principles of Programming Languages (POPL), pp. 301–314 (2013)
5. Myreen, M.O.: Formal Verification of Machine-Code Programs. PhD thesis, University of Cambridge (2009)
6. Nipkow, T.: Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. Formal Aspects of Computing 10, 171–186 (1998)
7. Norrish, M.: Mechanised Computability Theory. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 297–311. Springer, Heidelberg (2011)
8. Post, E.: Finite Combinatory Processes-Formulation 1. Journal of Symbolic Logic 1(3), 103–105 (1936)
9. Robinson, R.M.: Undecidability and Nonperiodicity for Tilings of the Plane. Inventiones Mathematicae 12, 177–209 (1971)
10. Zammit, V.: On the Readability of Machine Checkable Formal Proofs. PhD thesis, University of Kent (1999)

A Machine-Checked Proof of the Odd Order Theorem

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry

Microsoft Research - Inria Joint Centre

Abstract. This paper reports on a six-year collaborative effort that culminated in a complete formalization of a proof of the Feit-Thompson Odd Order Theorem in the Coq proof assistant. The formalized proof is constructive, and relies on nothing but the axioms and rules of the foundational framework implemented by Coq. To support the formalization, we developed a comprehensive set of reusable libraries of formalized mathematics, including results in finite group theory, linear algebra, Galois theory, and the theories of the real and complex algebraic numbers.

1 Introduction

The Odd Order Theorem asserts that every finite group of odd order is solvable. This was conjectured by Burnside in 1911 [34] and proved by Feit and Thompson in 1963 [14], with a proof that filled an entire issue of the Pacific Journal of Mathematics. The result is a milestone in the classification of finite simple groups, and was one of the longest proof to have appeared in the mathematical literature to that point. Subsequent work in the group theory community aimed at simplifying and clarifying the argument resulted in a more streamlined version of the proof, described in two volumes [6, 36]. The first of these, by H. Bender and G. Glauberman, deals with the "local analysis", a term coined by Thompson in his dissertation, which involves studying the structure of a group by focusing on certain subgroups. The second of these, by T. Peterfalvi, invokes character theory, a more "global" approach that involves studying a group in terms of the ways it can be represented as a group of matrices.

Both the size of this proof and the range of mathematics involved make formalization a formidable task. The last couple of decades have brought substantial advances in the use of interactive theorem provers, or "proof assistants," towards verifying substantial mathematical results [5, 16, 23]. The technology has also been used to verify the correctness of hardware and software components with respect to given specifications; significant successes in that area include the formal proof of correctness of a realistic compiler [32] and of a small operating system [29]. Formal methods have been especially useful when it comes to verifying the correctness of mathematical proofs that rely on computations that are too long to be checked

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 163–179, 2013.

by hand. For example, Appel and Haken's proof of the four-color theorem [1] has been verified [16] in the Coq system [7], and Thomas Hales' *Flyspeck* project [23] is working towards verifying a proof of the Kepler conjecture, which Hales himself first established with contributions by Samuel Ferguson.

The formalization described in the present article, however, is of a different nature. The proof of the Odd Order Theorem does not rely on mechanical computation, and the arguments were meant to be read and understood in their entirety. What makes the formalization difficult — and interesting — is the combination of theories involved. Working with these theories formally required developing a methodology that makes it possible to switch, efficiently, between the various views a mathematical text can superimpose on the same mathematical object. Another important task has been to formalize common patterns of mathematical reasoning. When it comes to formal verification of software, interaction with a proof assistant is commonly based on case analysis and structural induction. In contrast, the proof of the Odd Order Theorem relies on a variety of argument patterns that require new kinds of support.

In Section 2 we outline the statement and proof of the Odd Order Theorem. Section 3 provides some examples of the design choices we adopted to represent mathematical concepts in the type theory underlying the Coq system. In Section 4 we review some examples of the techniques we used to represent efficiently different kinds of proof patterns encountered in this proof. In Section 5 we provide three examples of advanced mathematical theories whose formalization require a robust combination of several areas of formalized mathematics, before scaling to the main proof. Section 6 concludes the paper with comments and some quantitative facts about this work.

2 An Overview of the Odd Order Theorem

2.1 Preliminaries

A *group* G consists of a set, usually also named G, together with an associative binary law $*$, usually denoted by juxtaposition, and an identity element 1, such that each element g of G has an inverse g^{-1}, satisfying $gg^{-1} = g^{-1}g = 1$. When there is no ambiguity, we identify an element g of a group with the corresponding singleton set $\{g\}$. In particular the trivial group $\{1\}$ is denoted by 1. The cardinality of G is called the *order* of the group. Examples of finite groups include the cyclic group $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n under addition, with identity 0; the set S_n of permutations of $\{0, \ldots, n-1\}$, under composition; and the set of isometries of a regular n-sided polygon. These examples have order n, $n!$, and $2n$, respectively. The cartesian product $G_1 \times G_2$ of two groups G_1 and G_2 is canonically a group with law $(a_1, a_2) * (b_1, b_2) := (a_1 b_1, a_2 b_2)$; the group $G_1 \times G_2$ is called the *direct product* of G_1 and G_2.

The law of an *abelian* group is commutative; in a non-abelian group G, we only have $ab = ba^b = ba[a, b]$, where $a^b := b^{-1}ab$ is the *b-conjugate* of a, and $[a, b] := a^{-1}b^{-1}ab$ is the *commutator* of a and b. Product and conjugation extend to subsets A, B of G, with $AB := \{ab \mid a \in A,\ b \in B\}$ and $A^b := \{a^b \mid a \in A\}$.

A subset A of G is *B-invariant* when $A^b = A$ for all b in B; in that case we have $AB = BA$.

One says that H is a *subgroup* of a group G, and writes $H < G$, when H is a subset of G containing 1 and closed under product and inverses — thus itself a group. For finite H, $H < G$ is equivalent to $1 \cup H^2 \subset H \subset G$. The set of subgroups of G is closed under intersection, conjugation, and commutative product (such as product with an invariant subgroup). If G is finite and $H < G$, then the order of H necessarily divides the order of G. It is not generally the case that for each divisor of the order of G there exists a subgroup of G of this order, but if G is a group of order n and p is a prime number dividing n with multiplicity k, then there exists a subgroup of G having order p^k, called a Sylow p-subgroup of G.

The notion of a *normal subgroup* is fundamental to group theory:

Definition 1 (Normal subgroup). *H is a* normal subgroup *of a group G, denoted $H \triangleleft G$, when H is a G-invariant subgroup of G.*

If $H \triangleleft G$, the set $\{Hg \mid g \in G\}$ of H-cosets is a group, as $(Hg_1)(Hg_2) = H(g_1g_2)$. This group, denoted G/H, is called the *quotient group* of G and H because it identifies elements of G that differ by an element of H. If G_1 and G_2 are groups, G_1 and G_2 are both normal in the group $G_1 \times G_2$.

Every finite abelian group is isomorphic to a direct product of cyclic groups $\mathbb{Z}/p_i^{k_i}\mathbb{Z}$, where the p_i are prime numbers. The far more complex structure of nonabelian groups can be apprehended using an analogue of the decomposition of a natural number by repeated division:

Definition 2 (Normal series, factors). *A* normal series *for a group G is a sequence $1 = G_0 \triangleleft G_1 \cdots \triangleleft G_n = G$, and the successive quotients $(G_{k+1}/G_k)_{0 \le k < n}$ are called the* factors *of the series.*

A group G is *simple* when its only proper normal subgroup is the trivial group 1, i.e., if its only proper normal series is $1 \triangleleft G$. A normal series whose factors are all simple groups is called a *composition series*. The Jordan-Hölder theorem states that the (simple) factors of a composition series play a role analogous to the prime factors of a number: two composition series of the same group have the same factors up to permutation and isomorphism. Unlike natural numbers, however, non-isomorphic groups may have composition series with isomorphic factors. The class of *solvable* groups is characterized by the elementary structure of their factors:

Definition 3 (Solvable group). *A group G is* solvable *if it has a normal series whose factors are all abelian.*

Subgroups and factors of solvable groups are solvable, so by the structure theorem for abelian groups, a finite group is solvable if and only if all the factors of its composition series are cyclic of prime order. We are now able to state the Odd Order Theorem.

2.2 The Odd Order Theorem

Theorem 1 (Odd Order theorem [14]). *Every finite group of odd order is solvable.*

It is striking that the theorem can be stated in such elementary terms, whereas its proof requires much more baggage. The file `stripped_Odd_Order`[1] of [39] provides a minimal, self-contained formulation of the Odd Order theorem, using only the bare Coq logic, and avoiding any use of extra-logical features such as notations, coercions or implicit arguments.

The proof of Theorem 1 proceeds by induction, showing that no minimal counterexample G exists. At the outset G is only known to be simple, nonabelian of odd order, but all proper subgroups of G should be solvable. The first half of the proof exploits these meager facts to derive a detailed description of the maximal proper subgroups of G, reducing the general structure of G to five cases. The second half of the proof uses character norm inequalities to rule out four of these, and extract some algebraic identities in a finite field from the last one. Galois theory is then used to refute these, completing the proof.

The study of the (solvable) subgroups of G exploits their decomposition into prime factors, reconstructing the structure of a maximal subgroup M from that of its p-factors for individual primes p. An A-invariant subgroup H of M has a normal series with A-invariant *elementary abelian* factors, that is, direct products of prime cycles. Identifying each one with a vector space over a finite field $\mathbf{F}_p$ makes it possible to analyze the action of A on H via the *representations* mapping A to a group of matrices over $\mathbf{F}_p$, and use linear algebra techniques such as eigenspace decomposition. Indeed, the proof starts by showing that 2×2 representations are abelian, then that no representation of A has a quadratic minimal polynomial (this replaces the use of the Hall-Higman theorem in [14]). This *p-stability* is combined with Glauberman's ZJ^* factorization to establish a Uniqueness theorem (Chapter II of [6]): any subgroup of *rank* 3 (containing an elementary abelian subgroup of order p^3) lies in a unique maximal subgroup of G.

Combining the Uniqueness theorem with results of Blackburn on odd groups of rank 2 yields that any maximal subgroup M of G is a *semidirect product* $M_\sigma E$ with $M_\sigma \triangleleft M$ and M_σ, E of coprime order. Furthermore, very few elements of M_σ and E commute — M is similar to a *Frobenius group*. Further analysis reveals that most M are of type I: M is very nearly a Frobenius group, with M_σ equal to the direct product M_F of the normal Sylow subgroups of M. However some M can be of type P, with $M = M_F U W_1$, where W_1 is cyclic, UW_1 is a Frobenius group, and all w_1 in W_1 commute precisely with the same cyclic group $W_2 < M_F$ (W_1 acts in a *prime manner* on M_F). Type P is subdivided into types V, II, III or IV, according to whether U is trivial, included in a different maximal group, abelian, or nonabelian, respectively. If any, there are exactly two type P groups up to conjugation, with W_1 and W_2 interchanged; at least one has type II, and over half of the elements of G lie in conjugates of $W = W_1 W_2$.

[1] `http://coqfinitgroup.gforge.inria.fr/doc/stripped_odd_order_theorem.html`

The second part of the proof [36] uses *characters*. The *character* of a complex representation $\rho : H \mapsto GL(n, \mathbb{C})$ is the function mapping each $h \in H$ to the trace of $\rho(h)$. In general, a character is *not* a group homomorphism, but it is a *class function*, constant on conjugacy classes of H. Convolution over H makes the set of class functions on a group H into a Hermitian space, for which the set irr H of *irreducible characters* of H forms an orthonormal basis. Characters of H have natural integer coordinates in irr H, hence integral norm.

Local analysis provides us both with a precise description of the characters of a maximal subgroup M, and an isometry mapping certain *virtual characters* of M (differences of characters) to virtual characters of G. This *Dade isometry* is only defined on functions that vanish on 1, so in order to extract usable information on G one needs *coherence* theorems extending it to a set of proper characters. The first, due to Sibley, covers Frobenius and type V maximal subgroups, and the second type II–IV subgroups.

For any $\chi \in$ irr G, coherence for a set (M_i) of non-conjugate maximal subgroups implies a numerical inequality bounding the sum of the (Hermitian) norms of the inverse images of the restrictions of χ to the support of the image of the Dade isometries for the M_i, and the (complex) norms of the values of χ elsewhere. For types III–V this bound yields a non-coherence theorem, which successively eliminates types V and IV; this implies that type I groups are actually Frobenius, and then the coherence bound forces type P groups to exist.

More inequalities then force the M_F, U, and W_1 subgroups of the type P groups to be isomorphic to the additive, unitary multiplicative, and Galois groups of a finite field $\mathbf{F}_{p^q}$ of order p^q, then rule out type III, and imply that U is W_2^y-invariant for some $y \in H_F$, where H is the *other* type II group such that $W_1 < H_F$. Intricate calculations show that this implies that if $a \in \mathbf{F}_{p^q}$ and $2 - a$ both have Galois norm 1, then so does $\tau(a) := 2 - 1/a$, and hence $\tau(a) \dots \tau^k(a) = (1 - 1/a)k + 1$; for $a \neq 1$ the Galois norm of $(1 - 1/a)x + 1$ yields a polynomial of degree q which has $0, ..., p - 1$ as roots, whence $q \leq p$ and hence $q = p$ by symmetry, so the orders of M_F and $E > W_1$ are not coprime, a contradiction.

2.3 Mathematical Sources

Our formalization follows the two books describing the revised proof [6, 36], with a small number of exceptions, for which we formalized the original arguments [14]. We followed Huppert's proof [27] of the Wielandt fixed point order formula. The elementary finite group theory part follows standard references [31]. For more advanced material we have used Aschbacher's and Gorenstein's books [3, 22] and some sources adapted to Galois theory and commutative algebra [30, 33, 37]. The formalization of character theory is based on Isaacs [28].

3 Mathematical Structures and Interfaces in Coq

3.1 The Calculus of Inductive Constructions and the Coq System

Most mathematical papers and textbooks do not explicitly specify a formal axiomatic foundation, but can generally be viewed as relying on set theory and

classical logic. Many interactive theorem provers, however, use alternative formal systems based on some form of type theory. Just as in the domain of programming languages, types help classify expressions passed to the checker, and hence facilitate the verification of claims in which they appear. The Coq proof assistant [7] is based on a logical foundation known as the *Calculus of Inductive Constructions*, or *CIC* [11, 12], a powerful and expressive version of constructive dependent type theory.

The advantage to using dependent type theory is that types can express complex specifications. For example, in our formalization, if G is an object of type `finGroupType`, then G is a record type which packages the type representing the elements of G, the binary group operation, the identity, and the inverse, as well as proof that these items satisfy the group axioms. In addition, Coq's type inference algorithm can eliminate the need to provide information that can be reconstructed from type constraints, just as implicit information is reconstructed by an experienced reader of a page of mathematics. For example, if g and h are elements of the carrier type of G an object of type `finGroupType`, then when presented with the expression $g * h$, Coq can infer that $*$ denotes the binary operation of G, as well as the fact that that operation is associative, and so on. Thus, type inference can be used to discover not only types, but also data and useful facts [19, 4]. Working with such an elaborate type system in a proof assistant can be delicate, however, and issues like the decidability of type checking impose severe restrictions on the nature of the dependent types one can work with in practice.

The status of computation in Coq's formalism also plays a central role in the present formalization. Every term or type has a computational interpretation, and the ability to unfold definitions and normalize expressions is built in to the underlying logic. Type inference and type checking can take advantage of this computational behavior, as can proof checking, which is just an instance of type checking. The price to pay for this powerful feature is that Coq's logic is constructive. In Coq many classical principles, such as the law of the excluded middle, the existence of choice functions, and extensionality are not available at the level of the logic. These principles can be recovered when they are provably valid, in the constructive sense, for specific objects like finite domains, or they can be postulated as axioms if needed. The present formalization, however, does not rely on any such axiom. Although it was not the primary motivation for this work, we eventually managed to obtain a completely constructive version of the proof and of the theories it requires.

In short, the success of such a large-scale formalization demands a careful choice of representations that are left implicit in the paper description. Taking advantage of Coq's type mechanisms and computational behavior allows us to organize the code in successive layers and interfaces. The lower-level libraries implement constructions of basic objects, constrained by the specifics of the constructive framework. Presented with these interfaces, the users of the higher-level libraries can then ignore these constructions, and they should hopefully be able to describe the proofs to be checked by Coq with the same level of comfort

as when writing a detailed page in LaTeX. The goal of this section is to describe some of the design choices that were made in that respect.

3.2 Principles of Boolean Reflection

The equality relation $x = y$ on a given type is generally not decidable, which is to say, the alternative $x = y \vee x \neq y$ is not generally provable. In some contexts, one can prove that equality is equivalent to a boolean relation $x \equiv y$, for which the law of the excluded middle holds. In our formalization, an `eqType` is a type paired with such a relation. Working with an `eqType` thus provides a measure of classical reasoning.

Using boolean values and operations to reason about propositions is called *boolean reflection*. This makes it possible, in a sense, to "calculate" with propositions, for example, by rewriting with boolean identities. More generally, to take advantage of propositions that can be represented as boolean values, the libraries provides infrastructure theorems to link logical connectives on (boolean) propositions with the corresponding boolean connectives. The SSReflect tactic language also provides support to facilitate going back and forth between different but equivalent descriptions of the same notion, like between boolean predicates and their logical equivalents. An explicit coercion is used throughout the formalization, which inserts automatically and silently an injection from a boolean value `b` to the formula `(b = true)` in the type `Prop` of propositions. This strategy is central to our methodology, and explains the name SSReflect, which is short for *small scale reflection* [18, 20].

Decidable equality plays another important role in facilitating the use of subtypes. If A is a type, a subset of A can be represented by a predicate B on A, that is, a map B from A to the type `Prop`. An element of this subset can be represented by an element a of A, and a "proof" p that B holds for a. The dependent pair $\langle a, p\rangle$ is an element of the *dependent sum*, or *Sigma type*, $\Sigma_{x:A} Bx$. As B takes values in `Prop`, $\Sigma_{x:A} Bx$ is also called a *subtype* of A, for the reasons just described.

The problem is that the equality on a subtype is not as simple as one would like. Two elements $\langle a_1, p_1\rangle$ and $\langle a_2, p_2\rangle$ are equal if and only if $a_1 = a_2$ but also, now considering both p_1 and p_2 as proofs that a_1 satisfies B, p_1 and p_2 are the *same* proof. However, a theorem due to Hedberg [25], formalized in our library as `eq_irrelevance`[2], implies that if B is a boolean-valued rather than `Prop`-valued predicate, then any two proofs p_1 and p_2 that a_1 satisfies B are equal. Thus, in this case, two elements $\langle a_1, p_1\rangle$ and $\langle a_2, p_2\rangle$ of the subtype are equal if and only if $a_1 = a_2$. Our libraries provide support [18] for the manipulations of these boolean subtypes, which are used pervasively in the formalization.

3.3 Finite Group Theory

Given the substantial amount of group theory that needed to be formalized, we relied on two important observations to optimize the data structures used to

[2] `http://coqfinitgroup.gforge.inria.fr/doc/eqtype.html`

represent finite groups. First, in many proofs, one can take most or all of the groups involved to be subgroups of a larger ambient group, sharing the same group operation and identity. Moreover, local notions like the normalizer of H inside of G, denoted $\mathrm{N}_G(H)$, are often used "globally," as in $\mathrm{N}(H)$, a practice which implicitly assumes that the normalizer is to be computed within an ambient container group. Second, and more importantly, many theorems of group theory are equally effective when stated in less generality, in terms of subgroups of such an ambient group. For example, given a theorem having to do with two unrelated groups, it does not hurt to assume that the two groups are subgroups of a larger group; this can always be made to hold by viewing them as subgroups of their direct product.

In our formalization, we represent such ambient groups as `finGroupType`s, and then represent the groups of interest as subsets of that type that contain the identity and are closed under the group operation. This is much simpler than having to maintain a plurality of types and morphisms between them. We form a new `finGroupType` only when strictly necessary, for example when forming a quotient group, which requires a new group operation [19].

A delicate point we had to cope with is that many constructions are partial. For example, the quotient group G/N can be formed only if $N \triangleleft G$ (see Section 2.1), and the direct product $G \times H$ of two subgroups of an ambient group can be formed only if they commute and $G \cap H = 1$. In addition, given our encoding of groups as subsets of a container group type, morphisms are unlikely to be defined on the whole group type, but rather on a specific subset.

Such constructions are ubiquitous. Forming subtypes as described in Section 3.2 in each case would be unwieldy and would make the application of lemmas mentioning partial constructions particularly cumbersome. The general approach has been to make each of these constructions total, either by modifying the input when it is invalid, or returning a default value. For example, the direct product construction returns the empty set (which is not a group) if the input groups have a nontrivial intersection; applying a morphism to a set automatically shrinks the input set by intersecting it with the domain of the morphism. The downside of this approach is that lemmas involving partial constructions often, but not always, depend on side conditions that need to be checked when the lemma is applied.

3.4 Dependent Records as First Class Interfaces

Records are just a generalization of the dependent pair construction described in Section 3.2, and, in the same way, can be used to package together types, data, operations and properties. They can thus play the role of abstract "interfaces." Such interfaces are very natural in abstract algebra, but are also useful in developing a theory of iterated operations [8], a theory of morphisms, a theory of algebraic structures [15] and so on. For an extensive list of interfaces used in the SSReflect library, the reader can refer to Section 11.3 of the SSReflect documentation [20].

In addition to the hierarchy of algebraic structures, we also provide a hierarchy for numeric fields [9], which are fields equipped with a boolean order relation, possibly partial. The purpose of this small hierarchy is to capture the operations and the theory of the complex number field and its subfields (cf Section 5.2).

Here we simply provide an example to illustrate how, using record types and setting up type inference carefully, one can obtain a hierarchy of interfaces that provides multiple inheritance, notation overloading, and (as we will see in the next section) a form of proof search. Consider the expression `(x + x * x == 0)`, where `x` is taken to be of type `int`. The symbols `*`, `+` and `==` are overloaded, and correspond to the multiplication of a ring, the addition operation in an additive group, and a decidable comparison operation. Type inference has to check that operations are applied to arguments of the right type; for example `x` of type `int` is used as an argument of `*`, hence the type `int` is *unified* with the carrier of an unspecified ring structure. Unification can be programmed, thanks to Coq's *canonical structures* mechanism, to solve such a unification problem by fixing the unknown structure to be the canonical ring structure on the integers. Given that `int` has been proved to be an instance of all the structures involved, unification always succeeds and type inference can make sense of the input expression, binding the overloaded symbols to the respective integer operations.

3.5 Searching Proofs by Programming Type Inference

Very often, the verification of small, uninteresting details is left implicit in an ordinary mathematical text, and it is assumed that a competent reader can fill these in. Canonical structures can be programmed to play a similar role. In particular, structures can package data with proofs, as in Section 3.2, in which case searching for a particular structure that contains a certain value can amount to looking for a proof that some property holds for that value.

A slight difficulty is that canonical structures are designed to guide unification, which is used by type inference to process *types* (like `int`), while here we need to process *values* (like the intersection of two sets). The crucial observation is that Coq's logic features dependent types, which means that values can be injected into types, even artificially, to make them available to unification, and hence to canonical structure resolution. We call the mechanism for doing this a *phantom type*. The use is similar to the use of phantom types in functional programming [26], where one enriches a type with a dummy (phantom) annotation to trick the type system into enforcing additional invariants.

Manifestations of this automatic proof search machinery are ubiquitous. For example, we can prove `(1 \in f @* (G :&: H))` by applying the `group1` lemma, which states that a set contains the unit element `1` if it happens to be a group. The canonical structure mechanism infers this group structure automatically for `f @* (G :&: H)`: if `G` and `H` have a group structure, then so does their intersection, as well as the image of that intersection under a group morphism `f`.

An advanced use of canonical structures is found in the definition of the `mxdirect` predicate, which states that its argument is a finite sum $\sum_{i=1}^{n} E_i$

of vector spaces $(E_i)_{i\in[1\ldots n]}$ that is moreover direct. What makes the `mxdirect` predicate unusual is that it is computed from the *syntax* of its argument, by comparing the rank $r(\sum_{i=1}^{n} E_i)$ of the vector space defined by the whole expression with the sum $\sum_{i=1}^{n} r(E_i)$ of the ranks of its components. If the two numbers are equal, the sum is direct. The canonical structure mechanism is programmed to recognize the syntax of an arbitrary sum of vector spaces, to collect the single spaces, to sum up their ranks. The `mxdirect` predicate is then defined as the comparison of the result with the rank of the whole initial expression [17].

The canonical structures inference mechanism essentially provides a Prolog like resolution engine that is used throughout the libraries to write statements that are more readable and easier to use. Programming this resolution engine can be quite tricky, and a more technical explanation would go beyond the scope of this paper [21].

4 Mathematical Proofs in Coq

4.1 Symmetries

One commonly invokes a symmetry argument in a mathematical proof by asserting that "without loss of generality" some extra assumption holds. For example, to prove a statement $P(x, y)$ in which x and y play symmetric roles, adding the assumption $x \leq y$ does not change the resulting theorem. Whereas an ordinary mathematical proof will leave it to the reader to infer the tacit argument, when doing formal proofs, it is useful to have support to fill in the details [24].

The SSReflect proof language provides a simple but effective tool in that regard: the `wlog` tactic [20]. This tactic performs a logical cut with a formula constructed from the *names* of the context items involved in the symmetry argument and the statement of the extra property the symmetry will exploit, both provided by the user. The logical cut generated by the proof shell involves the selected piece of context and the ongoing, usually very large, goal. For instance, when attempting to prove the statement `(P a b)`, the command `wlog H: a b / a <= b`, generates a first subgoal requiring a proof that `(H : forall x y, x <= y -> P x y)` holds, and another one to prove `(P a b)` under the assumption of `H`, which boils down to two applications of `H` if the statement `(P a b)` is actually symmetric in `a` and `b`. This simple tool has been instrumental at several places of the formalization, especially in large proofs using character theory [36].

4.2 Cycles of Inequalities

A standard pattern of reasoning seems to conclude out of blue that some assertion holds, from a proof that a chain of nonstrict *inequalities* in which the first and last terms are the same. The implicit content is a three-step proof: the circularity of the chain forces each inequality to be an equality; for each inequality, the equality case is characterized by a certain condition; hence the conjunction of these conditions holds and the desired statement follows from these. Typical

examples of such inequalities come from the properties of convex functions, e.g., the inequality between the arithmetic and geometric means is related to the strict convexity of the exponential function. The equality conditions can, however, be more elaborate; for example, the rank of a sum of finite dimensional vector spaces is smaller than the sum of the ranks of the summed vector spaces and equality holds if and only if the sum is direct.

In order to formalize this kind of proof efficiently, the SSReflect library uses notation to pair an inequality with the condition under which equality holds. For example, consider the following lemma:

```
Lemma nat_Cauchy m n : 2 * (m * n)<= m ^ 2 + n ^ 2 ?= iff (m == n)
```

This hides the conjunction of the following statements:

```
2 * (m * n) <= m ^ 2 + n ^ 2
(2 * (m * n) == m ^ 2 + n ^ 2) = (m == n)
```

The second statement is an equality reflecting the equivalence of the two boolean statements. Because the `rewrite` tactic can take multi-rules as arguments [20], rewriting with `nat_Cauchy` can affect several kinds of comparisons. The library provides support for using these statements, including the transitivity lemma collecting the equality conditions that is instrumental in capturing the pattern of reasoning described above. This technique has been a key ingredient in the formalization of advanced results[3] using character theory [36].

4.3 Proof Search by Large-Scale Reflection

Most of the proofs we worked from were not amenable to automation. A notable exception is found in Section 3 of the second volume of the proof [36], which deals with an indexed family of *virtual characters* (β_{ij}). These are defined to be integer linear combinations $\beta_{ij} = \sum_k z_k \chi_k$ of irreducible characters, where the collection of irreducible characters (χ_k) form a family of class functions that is orthonormal for the inner product $\langle \cdot, \cdot \rangle$. The array of virtual characters in question satisfies certain combinatorial constraints:

- For each i, $\langle \beta_{ij}, \beta_{ij} \rangle = 3$.
- For any two distinct elements on the same row or the same column ($i = i'$ or $j = j'$ but not both), $\langle \beta_{ij}, \beta_{i'j'} \rangle = 1$.
- Any two elements on distinct rows and columns ($i \neq i'$ and $j \neq j'$) are orthogonal, that is, $\langle \beta_{ij}, \beta_{i'j'} \rangle = 0$.

These conditions impose tight constraints on the β_{ij}'s.

A two-page combinatorial argument [36] shows that if there are at least four rows and two columns, then elements of the same column have a common irreducible character, and the respective coefficients are equal. We initially formalized this argument by hand. Later, Pascal Fontaine (one of the developers of the SMT solver veriT), provided us with an encoding[4] that made it possible

[3] See, for instance, `http://coqfinitgroup.gforge.inria.fr/doc/PFsection9.html`

[4] The SmtLib file is available at `http://coqfinitgroup.gforge.inria.fr/smt/th3_5.smt`

to automate the proof using a trusted connection between Coq and an SMT solver [2]. In order to have an automated version of the proof within the Coq system, we ultimately encoded the proof search directly, taking advantage of the symmetry of the problem. This was done using large-scale reflection, supported by notation-based reification. This automated version is shorter than our initial version, compiles twice as fast, and is intellectually more satisfying, as it eliminates unnecessary steps from the original proof [36].

4.4 Classical Reasoning

The instances of the mathematical structures of our hierarchy (see Section 3.4) are required to have boolean operators for the comparison and, possibly, the ordering of their inhabitants. We provide instances for all these structures, and all the instances needed for this formalization are in fact either finite types or countable types. We therefore benefit from other classical properties otherwise not available in the constructive logic of Coq. Indeed, countable types satisfy the functional choice axiom for boolean predicates (Markov's principle); functions on finite types can be represented by their graphs, which are extensional: the graphs of any two functions that are pointwise equal are in fact equal [18].

Boolean reflection extends to any first-order theory that has a decision procedure. In particular, the algebraic hierarchy mentioned in Section 3.4 has an interface for fields with a decidable first-order theory. The specification of this property uses a deep embedding of first-order formulas together with a boolean satisfiability predicate. Finite fields are of course instances of this interface, as are algebraically closed fields, which enjoy quantifier elimination [38, 10]. Decidable fields are used in the formalization of representation theory, both in dealing with modular representations, which are based on finite fields, and complex representations, which are based on algebraic complex numbers.

In other cases first-order decidability fails, notably for the rationals and for number fields. As a result, we elected not to rely on this interface for some basic results in the theory of group modules that cannot be proved constructively. Instead, we proved their double negation, expressed using the `classically` monadic operator [17, 35]:

```
Definition classically (P : Prop) : Prop :=
 forall b : bool, (P -> b) -> b.
```

Note the implicit use in this statement of the coercion mentioned in Section 3.2. The statement `(classically P)` is logically equivalent to `(~~ P)`, but this formulation is more useful in practice, because when using a hypothesis of the form `(classically P)` in the proof of a statement expressed as a boolean (on which excluded middle holds), one can constructively assume that `P` itself holds.

The `classically` operator is used only in the file formalizing the theory of group modules for representations. Note that although we use the `classically` operator to weaken the statement of some theorems we formalized, we did not need to alter the statement of Odd Order Theorem to describe its proof completely within the calculus of inductive constructions.

5 Mathematical Theories

5.1 Representations and Characters

Our treatment of linear algebra is organized in two levels. On the abstract level, a hierarchy of structures (see Section 3.4) provides interfaces, notations and shared theories for vectors, F-algebras and their morphisms. On the concrete level, these structures are instantiated by particular models, centered on matrices [17]. The central ingredient to this latter formalization is an extended Gaussian elimination procedure similar to LUP decomposition. This formalization of matrix algebra itself contains proofs of nontrivial results, covering determinants, Laplace expansion for cofactors, and properties of direct sums.

The choices we have made are validated by the successful formalization of representation theory, which depends on both finite group theory and linear algebra. Thanks to the underlying formalization of linear algebra in terms of concrete matrices, it is fairly easy to define the representations of a given group G in terms of square matrices with coefficients in a given field F, as well as other important notions, like the enveloping algebra of a representation, or a group module. Part of the theory of group modules requires the extra assumption that F has a decidable first-order theory. The library includes formal proofs of the fundamental results of representation theory, including Schur's lemma, Maschke's theorem, the Jacobson density theorem, the Jordan-Hölder theorem, Clifford's theorem, the Wedderburn structure theorem for semisimple rings, etc.

The next step is the finite group character theory, the main prerequisite for the second part of the proof [36]. Characters are defined as class functions with complex values, equipped with their standard convolution product. We first define the tuple of class functions on a given group G that are irreducible characters of G; then characters are class functions that are linear combinations of irreducible characters with natural number coordinates. Finally, we define virtual characters: class functions that are integer linear combinations of a given list of class functions. All these definitions are constructive, thanks to the finiteness of the group, the decidability of the first order theory of the coefficient field (Section 4.4) (here the complex algebraic numbers) and the Smith normal form for integer matrices. The formalization includes results like the theory of inertia groups, Burnside's $p^a q^b$ theorem, and Burnside's vanishing theorem [28].

5.2 Complex Algebraic Numbers

Our formalization of the character theory used in the second volume of the proof [36] is parametrized by a decidable field of complex numbers. In order to provide a concrete instance of this interface, we formalized a construction of the algebraic numbers. Standard presentations of character theory use arbitrary complex numbers, but as characters can only take algebraic values, the restriction is innocuous.

The algebraics can be described as an algebraic closure of the rationals equipped with an involutive conjugation automorphism $z \mapsto \bar{z}$. The latter yields both a norm ($|z| = t\bar{t}$ for some $t^2 = z$) and a partial order ($x \leq y$ if $|y - x| = y - x$)

whose restriction to the real (conjugation-invariant) algebraics is total; that is, an implementation of our "numeric field" interface (Section 3.4).

We first obtained the algebraics as the complex extension $R[i]$ of the real closed field of real algebraic numbers R which we had constructed explicitly [9]; we adapted an elementary proof of the Fundamental Theorem of Algebra (FTA) based on matrix algebra [13] to show that $R[i]$ is algebraically closed.

We then refactored that construction to eliminate the use of the real algebraics. We construct the algebraics directly as a countable algebraic closure, then construct conjugation by selecting a maximal real subfield. Because we are within a closure we can use Galois theory and adapt the usual proof of the FTA to show that conjugation is total.

5.3 Galois Theory

Galois theory establishes a link between field extensions and groups of automorphisms. A field extension is built by extending a base field with roots of polynomials that are irreducible on this base field. The vector space structure of such an extension plays an important role. The remarks and methods described in Section 3.3 apply in this situation: instead of assigning a type to each new extension, field extensions are formalized as intermediate fields between a fixed base field F and a fixed ambient splitting field extension L. A splitting field extension of F is a field extension of F generated by an explicit finite list of *all* the roots of a given polynomial.

All the constructions of this formalized Galois theory hence apply to extensions of a field F that are subfields of a field L. If K and E are intermediate extensions between F and L, the Galois group type (see Section 3.3) of E is the type of automorphisms of E. Then the Galois group $\mathrm{Gal}(E/K)$ of a field extension E/K is the set of automorphisms of the Galois group type that fix K. Partiality issues are dealt with in a manner similar to their treatment in finite group theory (see Section 3.3): the definitions take as arguments subspaces of the ambient field, but the theory is available for those vector spaces that are fields, a fact that can generally be inferred via a canonical structure. For example, $\mathrm{Gal}(E/K)$ is a set when E and K are vector spaces, but is equipped with a group structure as soon as E is a field.

It is interesting to note that standard Galois theory is usually carried out on normal extensions rather than on splitting fields. While the two notions are constructively equivalent, splitting fields are much easier to construct in practice.

6 Conclusion

The success of the present formalization relies on a heavy use of the inductive types [12] provided by Coq and on various flavors of reflection techniques. A crucial ingredient was the transfer of the methodology of "generic programming" to formal proofs, using the type inference mechanisms of the Coq system.

Our development includes more than 150,000 lines of proof scripts, including roughly 4,000 definitions and 13,000 theorems. The roughly 250 pages of

mathematics in our two main sources [6, 36] translate to about 40,000 lines of formal proof, which amounts to 4-5 lines of SSReflect code per line of informal text. During the formalization, we had to correct or rephrase a few arguments in the texts we were following, but the most time-consuming part of the project involved getting the base and intermediate libraries right. This required systematic consolidation phases performed after the production of new material. The corpus of mathematical theories preliminary to the actual proof of the Odd Order theorem represents the main reusable part of this work, and contributes to almost 80 percent of the total length. Of course, the success of such a large formalization, involving several people at different locations, required a very strict discipline, with uniform naming conventions, synchronization of parallel developments, refactoring, and benchmarking for synchronization with Coq.

As we have tried to make clear in this paper, when it comes to formalizing this amount of mathematics, there is no silver bullet. But the combined success of the many techniques we have developed shows that we are now ready for theorem proving in the large. The outcome is not only a proof of the Odd Order Theorem, but also, more importantly, a substantial library of mathematical components, and a tried and tested methodology that will support future formalization efforts.

Acknowledgments. The authors would like to thank the Coq team for their continuous development, improvement and maintenance of the Coq proof assistant.

References

[1] Appel, K., Haken, W.: Every map is four colourable. Bulletin of the American Mathematial Society 82, 711–712 (1976)

[2] Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011)

[3] Aschbacher, M.: Finite Group Theory. Cambridge Studies in Advanced Mathematics. Cambridge University Press (2000)

[4] Avigad, J.: Type inference in mathematics. Bulletin of the European Association for Theoretical Computer Science, EATCS (106), 78–98 (2012)

[5] Avigad, J., Harrison, J.: Formally verified mathematics. To appear in the Communications of the ACM

[6] Bender, H., Glauberman, G.: Local analysis for the Odd Order Theorem. Number 188 in London Mathematical Society, LNS. Cambridge University Press (1994)

[7] Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions. Springer, Berlin (2004)

[8] 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)

[9] Cohen, C.: Construction of real algebraic numbers in coq. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 67–82. Springer, Heidelberg (2012)

[10] Cohen, C., Mahboubi, A.: A formal quantifier elimination for algebraically closed fields. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 189–203. Springer, Heidelberg (2010)
[11] Coquand, T., Huet, G.: The calculus of constructions. Information and Computation 76(2-3), 95–120 (1988)
[12] Coquand, T., Paulin-Mohring, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg (1990)
[13] Derksen, H.: The fundamental theorem of algebra and linear algebra. American Mathematical Monthly 100(7), 620–623 (2003)
[14] Feit, W., Thompson, J.G.: Solvability of groups of odd order. Pacific Journal of Mathematics 13(3), 775–1029 (1963)
[15] 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)
[16] Gonthier, G.: Formal proof—the Four Color Theorem. Notices of the AMS 55(11), 1382–1393 (2008)
[17] Gonthier, G.: Point-free, set-free concrete linear algebra. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 103–118. Springer, Heidelberg (2011)
[18] Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. Journal of Formalized Reasoning 3(2), 95–152 (2010)
[19] 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)
[20] Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA (2012)
[21] Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In: ICFP, pp. 163–175 (2011)
[22] Gorenstein, D.: Finite Groups. AMS Chelsea Publishing Series (2007)
[23] Hales, T.: Formal proof. Notices of the AMS 55(11), 1370–1380 (2008)
[24] Harrison, J.: Without Loss of Generality. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 43–59. Springer, Heidelberg (2009)
[25] Hedberg, M.: A coherence theorem for Martin-Löf's type theory. Journal of Functional Programming 8(4), 413–436 (1998)
[26] Hinze, R.: Fun with phantom types. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, Cornerstones of Computing, pp. 245–262 (2003)
[27] Huppert, B., Blackburn, N.: Finite Groups II. Grundlehren Der Mathematischen Wissenschaften. Springer London, Limited (1982)
[28] Isaacs, I.: Character Theory of Finite Groups. AMS Chelsea Pub. Series (1976)
[29] Klein, G., et al.: sel4: Formal verification of an os kernel. In: SOPS ACM SIGOPS, pp. 207–220 (2009)
[30] Konrad, K.: Separability II, `http://www.math.uconn.edu/ kconrad/blurbs/galoistheory/separable2.pdf`
[31] Kurzweil, H., Stellmacher, B.: The Theory of Finite Groups: An Introduction. Universitext Series. Springer (2010)
[32] Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL, pp. 42–54. ACM Press (2006)

[33] Mines, R., Richman, F., Ruitenburg, W.: A course in constructive algebra. Universitext (1979). Springer (1988)
[34] Neumann, P.M., Mann, A.J.S., Tompson, J.C.: The collected papers of William Burnside, vol. I. Oxford University Press (2004)
[35] O'Connor, R.: Classical mathematics for a constructive world. Mathematical Structures in Computer Science 21, 861–882 (2011)
[36] Peterfalvi, T.: Character Theory for the Odd Order Theorem. London Mathematical Society, LNS, vol. 272. Cambridge University Press (2000)
[37] Rotman, J.: Galois Theory. Universitext (1979). Springer (1998)
[38] Tarski, A.: A Decision Method for Elementary Algebra and Geometry (1948); 2nd edn. University of California Press, Berkeley (1951)
[39] The Mathematical Component Team. A Formalization of the Odd Order Theorem using the Coq proof assistant (September 2012), `http://www.msr-inria.inria.fr/Projects/math-components/feit-thompson`

Kleene Algebra with Tests and Coq Tools for while Programs

Damien Pous*

CNRS – LIP, ENS Lyon, UMR 5668

Abstract. We present a Coq library about Kleene algebra with tests, including a proof of their completeness over the appropriate notion of languages, a decision procedure for their equational theory, and tools for exploiting hypotheses of a certain kind in such a theory.

Kleene algebra with tests make it possible to represent if-then-else statements and while loops in imperative programming languages. They were actually introduced as an alternative to propositional Hoare logic.

We show how to exploit the corresponding Coq tools in the context of program verification by proving equivalences of while programs, correctness of some standard compiler optimisations, Hoare rules for partial correctness, and a particularly challenging equivalence of flowchart schemes.

Introduction

Kleene algebra with tests (KAT) have been introduced by Kozen [19], as an equational system for program verification. A Kleene algebra with tests is a Kleene algebra (KA) with an embedded Boolean algebra of tests. The Kleene algebra component deals with the control-flow graph of the programs—sequential composition, iteration, and branching—while the Boolean algebra component deals with the conditions appearing in if-then-else statements, while loops, or pre- and post-assertions. This formalism is both concise and expressive, which allowed Kozen and others to give detailed paper proofs about various problems in program verification (see, e.g., [3,19,21,23]). More importantly, the equational theory of KAT is decidable and complete over relational models [24], and hypotheses of a certain kind can be eliminated [11, 15]. This suggests that a proof using KAT should not be done manually, but with the help of a computer. The goal of the present work is to give this possibility, inside the Coq proof assistant.

The underlying decision procedure cannot be formulated, a priori, as a simple rewriting system: it involves automata algorithms, it cannot be defined in Ltac, at the meta-level, and it does not produce a certificate which could easily be checked in Coq, a posteriori. This leaves us with only one possibility: defining a reflexive tactic [1, 8, 14]. Doing so is quite challenging: we basically have to prove completeness of KAT axioms w.r.t. the model of guarded string languages (the

* Work partially funded by the PiCoq project, ANR-10-BLAN-0305.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 180–196, 2013.

natural generalisation of languages for KA, to KAT), and to provide a provably correct algorithm for language equivalence of KAT expressions.

The completeness theorem is far from trivial; we actually have to formalise a lot of preliminary material: finite sums, finite sets, unique decomposition of Boolean expressions into sums of atoms, regular expression derivatives, expansion theorem for regular expressions, matrices, automata... As a consequence, we only give here a high-level overview of the involved mathematics, leaving aside standard definitions, technical details, or secondary formalisation tricks. The interested reader can consult the library, which is documented [30].

This work is a natural continuation of our previous work on KA [9]; the whole development [30] was however restarted from scratch: the few parts that could have been reused were rewritten more uniformly, and simplified a lot (e.g., construction of matrices, and KA completeness proof).

Outline. We first present KAT and its models (§1). We then sketch the completeness proof (§2), the decision procedure (§3), and the method used to eliminate hypotheses (§4). We finally illustrate the benefits of our tactics on several case-studies (§5), before discussing related works (§6), and concluding (§7).

1 Kleene Algebra with Tests

A Kleene algebra with tests consists of:

- a Kleene algebra $\langle X, \cdot, +, ^\star, 1, 0\rangle$ [18], i.e., an idempotent semiring with a unary operation, called "Kleene star", satisfying an axiom: $1 + x \cdot x^\star \leq x^\star$ and two inference rules: $y \cdot x \leq x$ entails $y^\star \cdot x \leq x$ and the symmetric one. (The preorder $(\leq)$ being defined by $x \leq y \triangleq x + y = y$.)
- a Boolean algebra $\langle B, \wedge, \vee, \neg, \top, \bot\rangle$;
- a homomorphism from $\langle B, \wedge, \vee, \top, \bot\rangle$ to $\langle X, \cdot, +, 1, 0\rangle$, that is, a function $[\cdot] : B \to X$ such that $[a \wedge b] = [a] \cdot [b]$, $[a \vee b] = [a] + [b]$, $[\top] = 1$, and $[\bot] = 0$.

The elements of the set B are called "tests"; we denote them by a, b. The elements of X, called "Kleene elements", are denoted by x, y, z. We usually omit the operator "$\cdot$" from expressions, writing xy for $x \cdot y$. The following (in)equations illustrate the kind of laws that hold in all Kleene algebra with tests:

$$[a \vee \neg a] = 1 \qquad [a \wedge (\neg a \vee b)] = [a][b] = [\neg(\neg a \vee \neg b)]$$
$$x^\star x^\star = x^\star \qquad (x + y)^\star = x^\star (y x^\star)^\star \qquad (x + xxy)^\star \leq (x + xy)^\star$$
$$[a]([\neg a]x)^\star = [a] \qquad [a]([a]x[\neg a] + [\neg a]y[a])^\star [a] \leq (xy)^\star$$

The laws from the first line come from the Boolean algebra structure, while the ones from the second line come from the Kleene algebra structure. The two laws from the last line are more interesting: their proof must mix both Boolean algebra and Kleene algebra reasoning. They are left to the reader as a non-trivial exercice; the tools we present in this paper allow one to prove them automatically.

1.1 The Model of Binary Relations

Binary relations form a Kleene algebra with tests; this is the main model we are interested in, in practice. The Kleene elements are the binary relations over a given set S, the tests are the predicates over this set, and the star of a relation is its reflexive transitive closure:

$$\begin{aligned} X &= \mathcal{P}(S \times S) & B &= \mathcal{P}(S) \\ x \cdot y &= \{(p,q) \mid \exists r, (p,r) \in x \wedge (r,q) \in y\} & a \wedge b &= a \cap b \\ x + y &= x \cup y & a \vee b &= a \cup b \\ x^\star &= \{(p_0,p_n) \mid \exists p_1 \dots p_{n-1}, \forall i < n, (p_i, p_{i+1}) \in x\} & \neg a &= S \setminus a \\ 1 &= \{(p,p) \mid p \in S\} & \top &= S \\ 0 &= \emptyset \qquad\qquad [a] = \{(p,p) \mid p \in a\} & \bot &= \emptyset \end{aligned}$$

The laws of a Kleene algebra are easily proved for these operations; note however that one needs either to restrict to decidable predicates (i.e., to take `S → bool` or {`p: S → Prop | forall p, S p ∨¬S p`} for B), or to assume the law of excluded middle: B must be a Boolean algebra, so that negation has to be an involution. This choice for B is left to the user of the library.

This relational model is typically used to interpret imperative programs: such programs are state transformers, i.e., binary relations between states, and the conditions appearing in these programs are just predicates on states. These conditions are usually decidable, so that the above constraint is actually natural.

The equational theory of Kleene algebra with tests is complete over the relational model [24]: any equation $x = y$ that holds universally in this model can be proved from the axioms of KAT. We do not need to formalise this theorem, but it is quite informative in practice: by contrapositive, if an equation cannot be proved from KAT, then it cannot be universally true on binary relations, meaning that proving its validity for a particular instantiation of the variables necessarily requires one to exploit additional properties of this particular instance.

1.2 Other Models

We describe two other models in the sequel: the syntactic model (§1.3) and the model of guarded string languages (§1.4); these models have to be formalised to build the reflexive tactic we aim at.

There are other important models of KAT. First of all, any Kleene algebra can be extended into a Kleene algebra with tests by embedding the two-element Boolean lattice. We also have traces models (where one keeps track of the whole execution traces of the programs rather than just their starting and ending points), matrices over a Kleene algebra with tests, but also models inherited from semirings like min-plus and max-plus algebra. The latter models have a degenerate Kleene star operation; they become useful when one constructs matrices over them, for instance to study shortest path algorithms.

Also note that like for Kleene algebra [9,20,29], KAT admits a natural "typed" generalisation, allowing for instance to encompass heterogeneous binary relations

and rectangular matrices. Our Coq library is actually based on this generalisation, and this deeply impacts the whole infrastructure; we however omit the corresponding details and technicalities here, for the sake of clarity.

1.3 KAT Expressions

Let p, q range over a set Σ of *letters* (or *actions*), and let $a_1, \ldots, a_n$ be the elements of a finite set Θ of *primitive tests*. *Boolean expressions* and *KAT expressions* are defined by the following syntax:

$$a, b ::= a_i \in \Theta \mid a \wedge a \mid a \vee a \mid \neg a \mid \top \mid \bot \qquad \text{(Boolean expressions)}$$
$$x, y ::= p \in \Sigma \mid [a] \mid x \cdot y \mid x + y \mid x^\star \mid 1 \mid 0 \ . \qquad \text{(KAT expressions)}$$

Given a Kleene algebra with tests $\mathcal{K} = \langle X, B, [\cdot] \rangle$, any pair of maps $\theta : \Theta \to B$ and $\sigma : \Sigma \to X$ gives rise to a KAT homomorphism allowing to interpret expressions in $\mathcal{K}$. Given two such expressions x and y, the equation $x = y$ is a *KAT theorem*, written KAT $\vdash x = y$, when the equation holds in any Kleene algebra with tests, under any interpretation. One checks easily that KAT expressions quotiented by the latter relation form a Kleene algebra with tests; this is the free Kleene algebra with tests over Σ and Θ. (We actually use this impredicative encoding of KAT derivability in the Coq library.)

1.4 Guarded Strings Languages

Guarded string languages are the natural generalisation of string languages for Kleene algebra with tests. We briefly define them.

An *atom* is a function from elementary tests (Θ) to Booleans; it indicates which of these tests are satisfied. We let α, β range over atoms, the set of which is denoted by At. (Technically, we represent elementary tests as finite ordinals of a given size n ($\Theta = \texttt{ord}\ n$), and we encode atoms as ordinals ($\texttt{At} = \texttt{ord}\ 2^n$). This allows us to avoid functional extensionality problems.) We let u, v range over *guarded strings*: alternating sequences of atoms and letters, which both start and end with an atom:

$$\alpha_1, p_1, \ldots, \alpha_n, p_n, \alpha_{n+1} \ .$$

The concatenation $u * v$ of two guarded strings u, v is a partial operation: it is defined only if the last atom of u is equal to the first atom of v; it consists in concatenating the two sequences and removing one copy of the shared atom in the middle.

The Kleene algebra with tests of guarded string languages is obtained by considering sets of guarded strings for X and sets of atoms for B:

$$X = \mathcal{P}\left((At \times \Sigma)^\star \times At\right) \qquad B = \mathcal{P}\left(At\right)$$
$$x \cdot y = \{u * v \mid u \in x \wedge v \in y\} \qquad a \wedge b = a \cap b$$
$$x + y = x \cup y \qquad a \vee b = a \cup b$$
$$x^\star = \{u_1 * \cdots * u_n \mid \exists u_1 \ldots u_n, \forall i \leq n, u_i \in x\} \qquad \neg a = At \setminus a$$
$$1 = \{\alpha \mid \alpha \in At\} \qquad \top = At$$
$$0 = \emptyset \qquad [a] = \{\alpha \mid \alpha \in a\} \qquad \bot = \emptyset$$

Note that we slightly abuse notation by letting α denote either an atom, or a guarded string reduced to an atom. Also note that the set $B = \mathcal{P}\left(At\right)$ has to be represented by the Coq type `At → bool`, to get a Boolean algebra on it.

2 Completeness

Let G be the unique homomorphism from KAT expressions to guarded string languages such that

$$G(a_i) = \{\alpha \mid \alpha(a_i) \text{ is true}\} \qquad G(p) = \{\alpha p \beta \mid \alpha, \beta \in At\}$$

Completeness of KAT over guarded string languages can be stated as follows.

Theorem 1. *For all KAT expressions x, y, $G(x) = G(y)$ entails* $\mathrm{KAT} \vdash x = y$.

This theorem is central to our development: it allows us to prove (in)equations in arbitrary models of KAT, by resorting to an algorithm deciding guarded string language equivalence (to be described in §3).

We closely follow Kozen and Smith' proof [24]. This proof relies on the completeness of Kleene algebra over languages, which we thus need to prove first.

2.1 Completeness of Kleene Algebra Axioms

Let R be the Kleene algebra homomorphism from regular expressions to (plain) string languages mapping a letter p to the language consisting of the single-letter word p. KA completeness over languages can be stated as follows [18]:

Theorem 2. *For all regular expressions x, y, $R(x) = R(y)$ entails* $\mathrm{KA} \vdash x = y$.

(Like for KAT, the judgement $\mathrm{KA} \vdash x = y$ means that $x = y$ holds in any Kleene algebra, under any interpretation.) We already presented a Coq formalisation of this theorem [9], but our development was over-complicated. We re-proved it from scratch here, following a simpler path which we now describe.

The main idea of Kozen's proof consists in replaying automata algorithms algebraically, using matrices to encode automata. The key insight that allowed us to considerably simplify the corresponding formalisation is that the algorithm used for this proof need not be the same as the one to be executed by the reflexive tactic we eventually define. Indeed, we can take the simplest possible algorithm to prove KA completeness, ignoring all complexity aspects, thus allowing us to

focus on conciseness and mathematical simplicity. In contrast, the algorithm to be executed by the final reflexive tactic should be relatively efficient, but we do not need to prove it complete, nor to replay its correctness algebraically: we only need to prove its correctness w.r.t. languages, which is much easier.

A preliminary step for the proof consists in proving that matrices over a Kleene algebra form a Kleene algebra. The Kleene star for matrices is non-trivial to define and to prove correct, but this can be done with a reasonable amount of efforts once appropriate lemmas and tools for block matrices have been set up.

A finite automaton can then be represented using three matrices (u, M, v) over regular expressions, where u is a $(1, n)$-matrix, M is a (n, n)-matrix, and v is a $(n, 1)$-matrix, n being the number of states of the automaton. Such a "matricial automaton" can be evaluated into a regular expression by taking the product $u \cdot M^\star \cdot v$, which is a scalar. The various classes of automata can be recovered by imposing conditions on the coefficients of the three matrices. For instance, a non-deterministic finite automaton (NFA) is such that u and v are 01-vectors and the coefficients of M are sums of letters.

Given a regular expression x, we construct a deterministic finite automaton (DFA) (u, M, v) such that $\mathrm{KA} \vdash x = uM^\star v$, as follows.

1. First construct a NFA with epsilon transitions (u'', M'', v''), such that $\mathrm{KA} \vdash x = u''M''^\star v''$. This is easily done by induction on x, using Thompson construction [31] (which is compositional, unlike the construction we used in [9]).
2. Remove epsilon transitions to obtain a NFA (u', M', v') such that $\mathrm{KA} \vdash u''M''^\star v'' = u'M'^\star v'$. We do it purely algebraically, in one line. In particular the transitive closure of epsilon transitions is computed using Kleene star on matrices. (Unlike in [9] we do not need a dedicated algorithm for this.)
3. Use the powerset construction to convert this NFA into a DFA (u, M, v) such that $\mathrm{KA} \vdash u'M'^\star v' = uM^\star v$. Again, this is done algebraically, and we do not need to perform the standard 'accessible subsets' optimisation.

We can prove that for any DFA (u, M, v), $R(uM^\star v)$ is the language recognised by the DFA. Therefore, to obtain Theorem 2, it suffices to prove that if two DFA (u, M, v) and (s, N, t) recognise the same language, then $\mathrm{KA} \vdash uM^\star v = sN^\star t$. For this last step, it suffices to exhibit a Boolean matrix that relates exactly those states of the two DFA that recognise the same language. We need for that an algorithm to check language equivalence of DFA states; we reduce the problem to DFA emptiness, and we perform a simple reachability analysis.

All in all, the KA completeness proof itself only requires 124 lines of specifications, and 119 lines of proofs (according to `coqwc`).

2.2 Completeness of KAT Axioms

To obtain KAT completeness (Theorem 1), Kozen and Smith [24] define a function $\widehat{\cdot}$ on KAT expressions that expands the expressions in such a way that we have $\mathrm{KAT} \vdash x = y$ iff $\mathrm{KA} \vdash \widehat{x} = \widehat{y}$. While this function can be thought as a reduction of KAT to KA, it cannot be used in practice: it produces expressions that are almost systematically exponentially larger than the given ones.

It is however sufficient to establish completeness; as explained earlier, we defer actual computations to a completely different algorithm (§3).

More precisely, the function $\widehat{\cdot}$ is defined in such a way that we have:

$$\mathrm{KAT} \vdash \widehat{x} = x \tag{i}$$
$$G(\widehat{x}) = R(\widehat{x}) \tag{ii}$$

We deduce KAT completeness as follows:

$$\begin{aligned}
& G(x) = G(y) && \\
\Leftrightarrow\ & G(\widehat{x}) = G(\widehat{y}) && (G \text{ is a KAT morphism, and (i)}) \\
\Leftrightarrow\ & R(\widehat{x}) = R(\widehat{y}) && (\text{by (ii)}) \\
\Rightarrow\ & \mathrm{KA} \vdash \widehat{x} = \widehat{y} && (\text{KA completeness}) \\
\Rightarrow\ & \mathrm{KAT} \vdash \widehat{x} = \widehat{y} && (\text{any KAT is a KA}) \\
\Leftrightarrow\ & \mathrm{KAT} \vdash x = y && (\text{by (i)})
\end{aligned}$$

(Note that the last equation entails the first one, so that all these statements are in fact equivalent.)

The function $\widehat{\cdot}$ is defined recursively over KAT expressions, using an intermediate datastructure: formal sums of *externally guarded terms* (i.e., either an atom, or a product of the form $\alpha x \beta$). The case of a starred expression $x^\star$ is quite involved: $\widehat{x^\star}$ is defined by an internal recursion on the length of the formal sum corresponding to $\widehat{x}$. The proof of the first equation (i) is not too difficult to formalise, using appropriate tools for finite sums (i.e., a simplified form of big operators [7], which we actually use a lot in the whole development). The second one (ii) is more cumbersome, notably because we must deal with the two implicit coercions appearing in its statement: formally, it has to be stated as follows:

$$i(G(\widehat{x})) = R(j(\widehat{x})) \ ,$$

where i takes a guarded string language and returns a finite word language on the alphabet $\Sigma \uplus \Theta \uplus \Theta$, and j takes a KAT expression and returns a regular expression over this extended alphabet, by pushing all negations to the leaves.

Apart from the properties of these coercion functions, the proof of (ii) mainly consists in rather technical arguments about regular and guarded string languages concatenation. All in all, once KA completeness has been proved, KAT completeness requires us 278 lines of specifications, and 360 lines of proofs.

3 Decision Procedure

To check whether two expressions denote the same language of guarded strings, we use an algorithm based on a notion of *partial derivatives* for KAT expressions. Derivatives were introduced by Brzozowski [10] for regular expressions; they make it possible to define a deterministic automaton where the states of the automaton are the regular expressions themselves.

$$\delta'_{\alpha,p}(x+y) = \delta'_{\alpha,p}(x) \cup \delta'_{\alpha,p}(y) \qquad \delta'_{\alpha,p}(q) = \begin{cases} \{1\} & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta'_{\alpha,p}(xy) = \begin{cases} \delta'_{\alpha,p}(x)y \cup \delta'_{\alpha,p}(y) & \text{if } \epsilon_\alpha(x) \\ \delta'_{\alpha,p}(x)y & \text{otherwise} \end{cases} \qquad \delta'_{\alpha,p}([a]) = \emptyset$$

$$\delta'_{\alpha,p}(x^\star) = \delta'_{\alpha,p}(x)x^\star$$

Fig. 1. Partial derivatives for KAT expressions

Derivatives can be extended to KAT expressions in a very natural way [22]: we first define a Boolean function ϵ_α, that indicates whether an expression accepts the single atom α; this function is then used to define the derivation function $\delta_{\alpha,p}$, that intuitively returns what remains of the given expression after reading the atom α and the letter p. These two functions make it possible to give a coalgebraic characterisation of the function G, which underpins the correctness of the algorithm we sketch below:

$$G(x)(\alpha) = \epsilon_\alpha(x) \qquad G(x)(\alpha\, p\, u) = G(\delta_{\alpha,p}(x))(u) \ .$$

Like with standard regular expressions, the set of derivatives of a given KAT expression (i.e., the set of expressions that can be obtained by repeatedly deriving w.r.t. arbitrary atoms and letters) can be infinite. To recover finiteness, we switch to *partial* derivatives [4]. Their generalisation to KAT should be folklore; we define them in Fig. 1. We use the notation Xy to denote the set $\{xy \mid x \in X\}$ when X is a set of expressions and y is an expression. The partial derivation function $\delta'_{\alpha,p}$ returns a (finite) set of expressions rather than a single one; this corresponds to the fact that we build a non-deterministic automaton. Still abusing notations, by letting a set of expressions denote the sum of its elements, we prove that $\mathrm{KAT} \vdash \delta_{\alpha,p}(x) = \delta'_{\alpha,p}(x)$.

Now call *bisimulation* any relation R between sets of expressions such that whenever $X\ R\ Y$, we have

- $\epsilon(X) = \epsilon(Y)$ and
- $\forall \alpha \in At, \forall p \in \Sigma,\ \delta'_{\alpha,p}(X)\ R\ \delta'_{\alpha,p}(Y)$.

We show that if there is a bisimulation R such that $X\ R\ Y$, then $G(X) = G(Y)$ (the converse also holds). This gives us an algorithm to decide language equivalence of two KAT expressions x, y: it suffices to try to construct a bisimulation that relates the singletons $\{x\}$ and $\{y\}$. This algorithm terminates because the set of partial derivatives reachable from a pair of expressions is finite (we do not need to formalise this fact since we just need the correctness of this algorithm).

There is a lot of room for optimisation in our implementation—for instance, we use unordered lists to represent binary relations. An important point in our design is that such optimisations can be introduced and proved correct independently from the completeness proof for KAT, which gives us much more flexibility than in our previous work on Kleene algebra [9].

3.1 Building a Reflexive Tactic

Using standard methodology [1, 8, 14], we finally pack the previous ingredients into a Coq reflexive tactic called `kat`, allowing us to close automatically any goal which belongs to the equational theory of KAT.

The tactic works on any model of KAT: those already declared in the library (relations, languages, matrices, traces), but also the ones declared by the user. The reification code is written in OCaml; it is quite complicated for at least two reasons: KAT is a two-sorted structure, and we actually deal with "typed" KAT, as explained in §1.2, which requires us to work with a dependently typed syntax.

For the sake of simplicity, the Coq algorithm we implemented for KAT does not produce a counter-example in case of failure. To be able to give such a counter-example to the user, we actually run an OCaml copy of the algorithm first (extracted from Coq, and modified by hand to produce counter-examples). This has two advantages: the tactic is faster in case of failure, and the counter-example—a guarded string—can be pretty-printed in a nicer way.

4 Eliminating Hypotheses

The above `kat` tactic works for the equational theory of KAT, i.e., the (in)equations that hold in any model of KAT, under any interpretation. In particular, this tactic does not make use of any hypothesis which is specific to the model or to the interpretation. Some hypotheses can however be exploited [11,15]: those having one of the following shapes.

(i) $x = 0$;
(ii) $[a]x = x[b]$, $[a]x \leq x[b]$, or $x[b] \leq [a]x$;
(iii) $x \leq [a]x$ or $x \leq x[a]$
(iv) $a = b$ or $a \leq b$;
(v) $[a]p = [a]$ or $p[a] = [a]$, for atomic p ($p \in \Sigma$);

Equations of the first kind (i) are called "Hoare" equations, for reasons to become apparent in §5.2. They can be eliminated using the following implication:

$$\begin{cases} x + uzu = y + uzu \\ z = 0 \end{cases} \quad \text{entails} \quad x = y \ . \tag{$\dagger$}$$

This implication is valid for any term u, and the method is complete [15] when u is taken to be the universal KAT expression, $\Sigma^\star$. Intuitively, for this choice of u, uzu recognizes all guarded strings that contain a guarded string of z as a substring. Therefore, when checking that $x + uzu = y + uzu$ are language equivalent rather than $x = y$, we rule out all counter-examples to $x = y$ that contain a substring belonging to z: such counter-examples are irrelevant since z is known to be empty.

Equations of the shape (iii) and (iv) are actually special cases of those of the shape (ii), which are in turn equivalent to Hoare equations. For instance, we have

$[a]x \leq x[b]$ iff $[a]x[\neg b] = 0$. Moreover, two hypotheses of shape (i) can be merged into a single one using the fact that $x = 0 \wedge y = 0$ iff $x + y = 0$. Therefore, we can aggregate all hypotheses of shape (i-iv) into a single one (of shape (i)), and use the above technique just once.

Hypotheses of shape (v) are handled differently, using the following equivalence:

$$[a]p = [a] \quad \text{iff} \quad p = [\neg a]p + [a] \ , \tag{‡}$$

This equivalence allows us to substitute $[\neg a]p + [a]$ for p in the considered goal—whence the need for p to be atomic. Again, the method is complete [15], i.e.,

$$\text{KAT} \vdash ([a]p = [a] \Rightarrow x = y) \quad \text{iff} \quad \text{KAT} \vdash x\theta = y\theta \qquad (\theta = \{p \mapsto [\neg a]p + [a]\})$$

4.1 Automating Elimination of Hypotheses in Coq

The previous techniques to eliminate some hypotheses in KAT can be easily automated in Coq. We first prove once and for all the appropriate equivalences and implications (the tactic `kat` is useful for that). We then define some tactics in Ltac that collect hypotheses of shape (i-iv), put them into shape (i), and aggregate them into a single one which is finally used to update the goal according to (†). Separately, we define a tactic that rewrites in the goal using all hypotheses of shape (v), through (‡). Finally, we obtain a tactic called `hkat`, that just preprocesses the conclusion of the goal using all hypotheses of shape (i-v) and then calls the `kat` tactic. Note that the completeness of this method [15] is a meta-theorem; we do not need to formalise it.

5 Case Studies

We now present some examples of Coq formalisations where one can take advantage of our library.

5.1 Bigstep Semantics of 'While' Programs

The bigstep semantics of 'while' programs is taught in almost every course on semantics and programming languages. Such programs can be embedded into KAT in a straightforward way [21], thus providing us with proper tools to reason about them. Let us formalise such a language in Coq.

Assume a type `state` of states, a type `loc` of memory locations, and an `update` function allowing to update the value of a memory location. Call *arithmetic expression* any function from states to natural numbers, and *Boolean expression* any function from states to Booleans (we use a partially shallow embedding). The 'while' programming language is defined by the inductive type below:

```
Variable loc, state: Set.
Variable update: loc → nat → state → state.

Definition expr := state → nat.
Definition test := state → bool.
```

```
Inductive prog :=
| skp
| aff (l: loc) (e: expr)
| seq (p q: prog)
| ite (b: test) (p q: prog)
| whl (b: test) (p: prog).
```

The bigstep semantics of such programs is given as a "state transformer", i.e., a binary relation between states. Following standard textbooks, one can define this semantics in Coq using an inductive predicate:

```
Inductive bstep: prog → rel state state :=
| s_skp: ∀ s, bstep skp s s
| s_aff: ∀ l e s, bstep (aff l e) s (update l (e s) s)
| s_seq: ∀ p q s s' s'', bstep p s s' → bstep q s' s'' → bstep (seq p q) s s''
| s_ite_ff: ∀ b p q s s', ¬ b s → bstep q s s' → bstep (ite b p q) s s'
| s_ite_tt: ∀ b p q s s', b s → bstep p s s' → bstep (ite b p q) s s'
| s_whl_ff: ∀ b p s, ¬ b s → bstep (whl b p) s s
| s_whl_tt: ∀ b p s s', b s → bstep (seq p (whl b p)) s s' → bstep (whl b p) s s'.
```

Alternatively, one can define this semantic through the relational model of KAT, by induction over the program structure:

```
Fixpoint bstep (p: prog): rel state state :=
  match p with
    | skp ⇒ 1
    | seq p q ⇒ bstep p·bstep q
    | aff l e ⇒ upd l e
    | ite b p q ⇒ [b]·bstep p + [¬b]·bstep q
    | whl b p ⇒ ([b]· bstep p)^*·[¬b]
  end.
```

(Notations come for free since binary relations are already declared as a model of KAT in our library.) The 'skip' instruction is interpreted as the identity relation; sequential composition is interpreted by relational composition. Assignments are interpreted using an auxiliary function, defined as follows:

```
Definition upd l e: rel state state := fun s s' ⇒ s' = update l (e s) s.
```

For the 'if-then-else' statement, the Boolean expression `b` is a predicate on states, i.e., a test in our relational model of KAT; this test is used to guard both branches of the possible execution paths. Accordingly for the 'while' loop, we iterate the body of the loop guarded by the test, using Kleene star. We make sure one cannot exit the loop before the condition gets false by post-guarding the iteration with the negation of this test.

This alternative definition is easily proved equivalent to the previous one. Its relative conciseness makes it easier to read (once one knows KAT notation); more importantly, this definition allows us to exploit all theorems and tactics about KAT, for free. For instance, suppose that one wants to prove some program equivalences. First define program equivalence, through the bigstep semantics:

```
Notation "p ∼ q" := (bstep p == bstep q).
```

(The "==" symbol denotes equality in the considered KAT model; in this case, relational equality.) The following lemmas about unfolding loops and dead code elimination, can be proved automatically.

```
Lemma two_loops b p: whl b (whl b p) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(([b]·bstep p)^*·[¬b]))^*·[¬b] == ([b]·bstep p)^*·[¬b] *)
```

```
Lemma fold_loop b p: whl b (p ; ite b p skp) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(bstep p·([b]·bstep p + [¬b]·1)))^*·[¬b] == ([b]·bstep p)^*·[¬b] *)

Lemma dead_code a b p q r:  whl (a ∨ b) p ; ite b q r  ∼  whl (a ∨ b) p ; r.
Proof. simpl. kat. Qed.
(* ([a ∨ b]·bstep p)^*·[¬(a ∨ b)]·([b]·bstep q + [¬b]·bstep r)
                          == ([a ∨ b]·bstep p)^*·[¬(a ∨ b)]·bstep r *)
```

(The semicolon in program expressions is a notation for sequential composition; the comments below each proof show the intermediate goal where the `bstep` fixpoint has been simplified, thus revealing the underlying KAT equality.)

Of course, the `kat` tactic cannot prove arbitrary program equivalences: the theory of KAT only deals with the control-flow graph of the programs and with the Boolean expressions, not with the concrete meaning of assignments or arithmetic expressions. We can however mix automatic steps with manual ones. Consider for instance the following example, where we prove that an assignment can be delayed. Our tactics cannot solve it automatically since some reasoning about assignments is required; however, by asserting manually a simple fact (in this case, an equation of shape (ii)), the goal becomes provable by the `hkat` tactic.

```
Definition subst l e (b: test): test := fun s ⇒ b (update l (e s) s).
Lemma aff_ite l e b p q: (l←e; ite b p q) ∼ (ite (subst l e b) (l←e; p) (l←e; q)).
Proof.
  simpl. (* upd l e·([b]·bstep p + [¬b]·bstep q) ==
       [subst l e b]·(upd l e·bstep p)·[¬subst l e b]·(upd l e·bstep q) *)
  assert (upd l e·[b] == [subst l e b]·upd l e) by (cbv; firstorder; subst; eauto).
  hkat.
Qed.
```

5.2 Hoare Logic for Partial Correctness

Hoare logic for partial correctness [16] is subsumed by KAT [21]. The key ingredient in Hoare logic is the notion of a "Hoare triple" $\{A\}\,p\,\{B\}$, where p is a program, and A, B are two formulas about the memory manipulated by the program, respectively called pre- and post-conditions. A Hoare triple $\{A\}\,p\,\{B\}$ is *valid* if whenever the program p starts in some state s satisfying A and terminates in a state s', then s' satisfies B. Such a statement can be translated into KAT as a simple equation:

$$[A]p[\neg B] = 0$$

Indeed, $[A]p[\neg B] = 0$ precisely means that there is no execution path along p that starts in A and ends in $\neg B$. Such equations are Hoare equations (they have the shape (i) from §4), so that they can be eliminated automatically. As a consequence, inference rules of Hoare logic can be proved automatically using the `hkat` tactic. For instance, for the 'while' rule, we get the following script:

```
Lemma rule_whl A b p: {A ∧ b} p {A} → {A} whl b p {A ∧ ¬b}.
Proof. simpl. hkat. Qed.
(* [A ∧ b]·bstep p·[¬A] == 0 → [A]·(([b]·bstep p)^⋆·[¬b])·[¬(A ∧ ¬b)] == 0 *)
```

5.3 Compiler Optimisations

Kozen and Patron [23] use KAT to verify a rather large range of standard compiler optimisations, by equational reasoning. Citing their abstract, they cover "*dead code elimination, common subexpression elimination, copy propagation, loop hoisting, induction variable elimination, instruction scheduling, algebraic simplification, loop unrolling, elimination of redundant instructions, array bounds check elimination, and introduction of sentinels*". They cannot use automation, so that the size of their proofs ranges from a few lines to half a page of KAT computations.

We formalised all those equational proofs using our library. Most of them can actually be solved instantaneously, by a simple call to the `hkat` tactic. For the few remaining ones, we gave three to four line proofs, consisting of first rewriting using hypotheses that cannot be eliminated, and then a call to `hkat`.

The reason why `hkat` performs so well is that most assumptions allowing to optimise the code in these examples are of the shape (i-v). For instance, to state that an instruction p has no effect when $[a]$ is satisfied, we use an assumption $[a]p = [a]$. Similarly, to state that the execution of a program x systematically enforces $[a]$, we use an assumption $x = x[a]$. The assumptions that cannot be eliminated are typically those of the shape $pq = qp$: "the instructions p and q commute"; such assumptions have to be used manually.

5.4 Flowchart Schemes

The last example we discuss here is due to Paterson, it consists in proving the equivalence of two flowchart schemes (i.e., goto programs—see Manna's book [26] for a complete description of this model). Manna proves their equivalence using several successive graph transormations. His proof is really high-level and informal; it is one page long, plus three additional pages to draw intermediate flowcharts schemes. Angus and Kozen [3] give a rather detailed equational proof in KAT, which is about six pages long. Using the `hkat` tactic together with some ad-hoc rewriting tools, we managed to formalise Angus and Kozen's proof in three rather sparse screens.

Like in Angus and Kozen's proof, we progressively modify the KAT expression corresponding to the first schema, to make it evolve towards the expression corresponding to the second schema. Our mechanised proof thus roughly consists in a sequence of transitivity steps closed by `hkat`, allowing us to perform some rewriting steps manually and to move to the next step. This is illustrated schematically by the code presented in Fig. 2.

Most of our transitivity steps (the y_i's) already appear in Angus and Kozen's proof; we can actually skip a lot of their steps, thanks to `hkat`. Some of these

```
Lemma Paterson: x_1 == z.
Proof.
  transitivity y_1. hkat.        (* x_1 == y_1 *)
  a few rewriting steps transforming y_1 into x_2.
  transitivity y_2. hkat.        (* x_2 == y_2 *)
  a few rewriting steps transforming y_2 into x_3.
  (* ... *)
  transitivity y_12. hkat.       (* x_12 == y_12 *)
  a few rewriting steps transforming y_12 into x_13.
  hkat.                          (* x_13 == z *)
Qed.
```

Fig. 2. Squeleton for the proof of equivalence of Paterson's flowchart schems

simplifications can be spectacular: for instance, they need one page to justify the passage between their expressions (24) and (27), while a simple call to `hkat` does the job; similarly for the page they need between their steps (38) and (43).

6 Related Works

Several formalisations of algorithms and results related to regular expressions and languages have been proposed since we released our Coq reflexive decision procedure for Kleene algebra [9]: partial derivatives for regular expressions [2], regular expression equivalence [6, 12, 25, 27], regular expression matching [17]. None of these works contains a formalised proof of completeness for Kleene algebra, so that they cannot be used to obtain a general tactic for KA (note however that Krauss and Nipkow [25] obtain an Isabelle/HOL tactic for binary relations using a nice trick to sidestep the completeness proof—but they cannot deal with other models of KA).

On the more algebraic side, Struth et al. [5, 13] showed how to formalise and use relation algebra and Kleene algebra in Isabelle/HOL; they exploit the automation tools provided by this assistant, but they do not try to define decision procedures specific to Kleene algebra, and they do not prove completeness.

To the best our knowledge, the only formalisation of KAT prior to the present work is due to Pereira and Moreira [28], in Coq. They state all axioms of KAT, derive some simple consequences of these axioms (e.g., Boolean disjunction distribute over conjunction, Kleene star is monotone), and use them to manually prove the inference rules of Hoare logic, as we did automatically in §5.2. They do not provide models, automation, decision procedure, or completeness proof.

7 Conclusion

We presented a rather exhaustive Coq formalisation of Kleene algebra with tests: axiomatisation, models, completeness proof, decision procedure, elimination of hypotheses. We then showed several use-cases for the corresponding library:

proofs about while programs and Hoare logic, certification of standard compiler optimisations, and equivalence of flowchart schemes.

Most of the theoretical material is due to Kozen et al. [3, 15, 18–24], so that our contribution mostly lies in the Coq mechanisation of these ideas. The completeness proof was particularly challenging to formalise, and lots of aspects of this work could not be explained in this extended abstract: how to encode the algebraic hierarchy, how to work efficiently with finite sets and finite sums, how to exploit symmetry arguments, reflexive normalisation tactics, tactics about lattices, finite ordinals and encodings of set-theoretic constructs in ordinals...

The Coq library is available online [30]; it is documented and axiom-free. This library actually has a larger scope than what we presented here: our long-term goal is to formalise and automate other fragments of relation algebra (residuated structures, Kleene algebra with converse, allegories...), so that the library is designed to allow for such extensions. For instance normalisation tactics and an ad-hoc semi-decision procedures are already defined for algebraic structures beyond Kleene algebra and KAT.

According to `coqwc`, the library consists of 4377 lines of specifications and 3020 lines of proofs, that distribute as follows. Overall, this is slightly less than our previous library for KA [9] (5105+4315 lines), and we do much more: not only we handle KAT, but we also lay the ground for the mechanisation of other fragments of relation algebra, as explained above.

	specifications	proofs	comments
ordinals, comparisons, finite sets...	674	323	225
algebraic hierarchy	490	374	216
models (languages, relations, expressions...)	1279	461	404
linear algebra, matrices	534	418	163
completeness, decisions procedure, tactics	1400	1444	740

The resulting theorems and tactics allowed us to shorten significantly a number of paper proofs—those about Hoare logic, compiler optimisations, and flowchart schemes. Getting a way to guarantee that such proofs are correct is important: although mathematically simple, they tend to be hard to proofread (we invite the skeptical reader to check Angus and Kozen's paper proof of Paterson example [3]). Moreover, automation greatly helps when searching for such proofs: being able to get either a proof or a counter-example for any proposed equation is a big plus: it makes it much easier to progress in the overall proof.

References

1. Allen, S.F., Constable, R.L., Howe, D.J., Aitken, W.E.: The semantics of reflected proof. In: Proc. LICS, pp. 95–105. IEEE Computer Society (1990)
2. Almeida, J.B., Moreira, N., Pereira, D., de Sousa, S.M.: Partial derivative automata formalized in Coq. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 59–68. Springer, Heidelberg (2011)
3. Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Technical Report TR2001-1844, CS Dpt, Cornell University (July 2001)

4. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. TCS 155(2), 291–319 (1996)
5. Armstrong, A., Struth, G.: Automated reasoning in higher-order regular algebra. In: Kahl, W., Griffin, T.G. (eds.) RAMICS 2012. LNCS, vol. 7560, pp. 66–81. Springer, Heidelberg (2012)
6. Asperti, A.: A compact proof of decidability for regular expression equivalence. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 283–298. Springer, Heidelberg (2012)
7. Bertot, Y., Gonthier, G., Biha, S.O., Pasca, I.: Canonical big operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 86–101. Springer, Heidelberg (2008)
8. Boyer, R., Moore, J.: Metafunctions: proving them correct and using them efficiently as new proof procedures. In: The Correctness Problem in Computer Science. Academic Press, NY (1981)
9. Braibant, T., Pous, D.: An efficient Coq tactic for deciding Kleene algebras. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 163–178. Springer, Heidelberg (2010)
10. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM 11(4), 481–494 (1964)
11. Cohen, E.: Hypotheses in Kleene algebra. Technical report, Bellcore, Morristown, N.J (1994)
12. Coquand, T., Siles, V.: A decision procedure for regular expression equivalence in type theory. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 119–134. Springer, Heidelberg (2011)
13. Foster, S., Struth, G.: Automated analysis of regular algebra. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 271–285. Springer, Heidelberg (2012)
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. Hardin, C., Kozen, D.: On the elimination of hypotheses in Kleene algebra with tests. Technical Report TR2002-1879, CS Dpt, Cornell University (October 2002)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
17. Komendantsky, V.: Reflexive toolbox for regular expression matching: verification of functional programs in Coq+ssreflect. In: Proc. PLPV, pp. 61–70. ACM (2012)
18. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. and Comp. 110(2), 366–390 (1994)
19. Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems 19(3), 427–443 (1997)
20. Kozen, D.: Typed Kleene algebra, TR98-1669, CS Dpt. Cornell University (1998)
21. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Log. 1(1), 60–76 (2000)
22. Kozen, D.: On the coalgebraic theory of Kleene algebra with tests. Technical Report CIS, Cornell University (March 2008), `http://hdl.handle.net/1813/10173`
23. Kozen, D., Patron, M.-C.: Certification of compiler optimizations using Kleene algebra with tests. In: Palamidessi, C., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 568–582. Springer, Heidelberg (2000)
24. 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)

25. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. JAR 49(1), 95–106 (2012)
26. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill (1974)
27. Moreira, N., Pereira, D., de Sousa, S.M.: Deciding regular expressions (In-) equivalence in Coq. In: Kahl, W., Griffin, T.G. (eds.) RAMICS 2012. LNCS, vol. 7560, pp. 98–113. Springer, Heidelberg (2012)
28. Pereira, D., Moreira, N.: KAT and PHL in Coq. Comput. Sci. Inf. Syst. 5(2), 137–160 (2008)
29. Pous, D.: Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 484–498. Springer, Heidelberg (2010)
30. Pous, D.: RelationAlgebra: Coq library containing all material presented in this paper (December 2012), `http://perso.ens-lyon.fr/damien.pous/ra`
31. Thompson, K.: Regular expression search algorithm. C. ACM 11, 419–422 (1968)

Program Analysis and Verification Based on Kleene Algebra in Isabelle/HOL

Alasdair Armstrong[1], Georg Struth[1], and Tjark Weber[2]

[1] Department of Computer Science, University of Sheffield, UK
{a.armstrong,g.struth}@dcs.shef.ac.uk
[2] Department of Information Technology, Uppsala University, Sweden
tjark.weber@it.uu.se

Abstract. Schematic Kleene algebra with tests (SKAT) supports the equational verification of flowchart scheme equivalence and captures simple while-programs with assignment statements. We formalise SKAT in Isabelle/HOL, using the quotient type package to reason equationally in this algebra. We apply this formalisation to a complex flowchart transformation proof from the literature. We extend SKAT with assertion statements and derive the inference rules of Hoare logic. We apply this extension in simple program verification examples and the derivation of additional Hoare-style rules. This shows that algebra can provide an abstract semantic layer from which different program analysis and verification tasks can be implemented in a simple lightweight way.

1 Introduction

The relevance of Kleene algebras for program development and verification has been highlighted for more than a decade. Kleene algebras provide operations for non-deterministic choice, sequential composition and finite iteration in computing systems as well as constructs for skip and abort. When a suitable boolean algebra for tests and assertions is embedded, the resulting Kleene algebras with tests (KAT) [18] can express simple while-programs and validity of Hoare triples. Extensions of Kleene algebras support Hoare-style program verification—the rules of Hoare logic except assignment can be derived—and provide notions of equivalence and refinement for program construction and transformation. Reasoning in Kleene algebras is based on first-order equational logic. It is therefore relatively simple, concise and well suited for automation [15,12,13,3]. The lightweight program semantics that Kleene algebras provide can further be specialised in various ways through their models, which include binary relations, program traces, paths in transition systems and (guarded string) languages [4].

The relevance of Kleene algebras has further been underpinned by applications, for instance, in compiler optimisation [19], program construction [5], transformation and termination [9], static analysis [11] or concurrency control [7]; but few have used theorem provers or integrated fine-grained reasoning about assignments or assertions [1,5,14]. The precise role and relevance of Kleene algebras

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 197–212, 2013.

in a formal environment for program development and verification has not yet been explored. Our paper provides a first step in this direction.

We have implemented a comprehensive library for KAT in Isabelle/HOL [25], using explicit carrier sets for modelling the interaction between actions and tests in programs. For reasoning about assignments and assertions, we have added first-order syntax and axioms to KAT, following Angus and Kozen's approach to schematic Kleene algebras with tests (SKAT) [2]. Program syntax is defined as syntactic sugar on SKAT expressions; axiomatic algebraic reasoning about these expressions is implemented by using Isabelle's quotient package [16,17].

We have applied this simple algebraic verification environment by formalising a complex flowchart equivalence proof in SKAT due to Angus and Kozen. It is an algebraic account of a previous diagrammatic proof by Manna [21]. In their approach, flowchart schemes are translated into SKAT expressions. We have converted the manual proof in SKAT essentially one-to-one into readable Isabelle code. This significantly shortens a previous formalisation with a customised interactive SKAT-prover [1]. This compression demonstrates the power of Isabelle's proof methods and integrated theorem provers.

To illustrate the flexibility of our approach we have extended our SKAT implementation by assertions for Hoare-style partial program correctness proofs. To obtain a predicate transformer semantics for forward reasoning à la Gordon [8] we have formalised the action of programs as SKAT terms which act on a Boolean algebra of predicates or assertions via a scalar product in a Kleene module [10,20]. We have instantiated this abstract algebra of assertions to the standard powerset algebra over program states realised as maps from variables to values. We have encoded validity of Hoare triples and automatically derived the rules of Hoare logic—including assignment—in this setting. We have also provided syntactic sugar for a simple while-language with assertions (pre/post-conditions and invariants) similar to existing Hoare logics in Isabelle [24,26].

We have tested this enhanced environment by automatically verifying some simple algorithms and by automatically deriving some additional Hoare-style inference rules that would be admissible in Hoare logic. Verification is supported by a verification condition generator that reduces program verification tasks to the usual proof obligations for elementary program actions.

The complete Isabelle code for this paper can be found online.[1]

Our study points out two main benefits of using (Kleene) algebra in program development and verification. First, it provides a uniform lightweight semantic layer from which syntax for specifications and programs can be defined, domain-specific inference rules be derived and fine-grained models be explored with exceptional ease. In Isabelle this is seamlessly supported by type classes and locales and by excellent proof automation. Second, it yields a powerful proof engine for concrete analysis tasks, in particular when transforming programs or developing them from specifications. Despite this, the automation of our flowchart example remains somewhat underwhelming; such examples provide interesting benchmarks for further improving proof automation.

[1] `http://www.dcs.shef.ac.uk/~alasdair/skat`

2 Kleene Algebra with Tests

Kleene algebras with tests (KAT) are at the basis of our implementation. They provide simple encodings of while-programs and Hoare logic (without assignment) and support equational reasoning about program transformations and equivalence. This section gives a short introduction from a programming perspective; more details can be found in the literature [18].

A *semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative semigroup, $(S, \cdot, 1)$ is a monoid (not necessarily commutative), multiplication distributes over addition from the left and right, and 0 is an annihilator ($0 \cdot p = 0 = p \cdot 0$). S is *idempotent* (a *dioid*) if $p + p = p$. In that case, the reduct $(S, +, 0)$ is a semilattice, hence $p \leq q \longleftrightarrow p + q = q$ defines a partial order with least element 0. For programming, imagine that S models the actions of a system; addition is non-deterministic choice, multiplication is sequential composition, 1 is skip and 0 is abort. The next step is to add a notion of finite iteration.

A *Kleene algebra* is a dioid expanded with a star operation that satisfies the unfold axioms $1 + pp^* \leq p^*$ and $1 + p^*p \leq p^*$, and the induction axioms $r + pq \leq q \longrightarrow p^*r \leq q$ and $r + qp \leq q \longrightarrow rp^* \leq q$. This defines p^* as the simultaneous least (pre)fixpoint of the functions $\lambda q.1 + pq$ and $\lambda q.1 + qp$.

Program tests and assertions can be added by embedding a boolean algebra of tests between 0 and 1. A *Kleene algebra with tests* (KAT) is a structure $(K, B, +, \cdot, ^\star, 0, 1, \overline{\ })$ where $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra and $(B, +, \cdot, \overline{\ }, 0, 1)$ a Boolean subalgebra of K. The operations are overloaded with $+$ as join, $\cdot$ as meet, 0 as the minimal element and 1 the maximal element of B. Complementation $\overline{\ }$ is only defined on B. We write p, q, r for arbitrary elements of K and a, b, c for tests in B. Conditionals and loops can now be expressed:

$$\mathit{IF}\ b\ \mathit{THEN}\ p\ \mathit{ELSE}\ q\ =\ bp + \overline{b}q, \qquad \mathit{WHILE}\ b\ \mathit{DO}\ p\ \mathit{WEND}\ =\ (bp)^*\overline{b}.$$

Tests play a double role as assertions to encode (the validity of) Hoare triples:

$$\{\!|b|\!\}\, p\, \{\!|c|\!\} \iff bp\overline{c} = 0.$$

Multiplying a program p by a test b at the left or right means restricting its input or output by the condition b. Thus the term $bp\overline{c}$ states that program p is restricted to precondition b in its input and to the negated postcondition c in its output. Accordingly, $bp\overline{c} = 0$ means that p cannot execute from b without establishing c. This faithfully captures the meaning of the Hoare triple $\{\!|b|\!\}\, p\, \{\!|c|\!\}$. It is well known that algebraic relatives of all rules of Hoare logic except assignment can be derived in KAT, and that binary relations under union, relational composition, the unit and the empty relation, and the reflexive transitive closure operation form a KAT. Its Boolean subalgebra of tests is formed by all elements between the empty and the diagonal relation. Binary relations yield, of course, a standard semantics for sequential programs.

A reference Isabelle implementation of Kleene algebras and their models is available in the Archive of Formal Proofs [4]. To capture the subalgebra relationship of B and K we have implemented an alternative with carrier sets and expanded this to KAT. Due to lack of space we cannot present further details.

3 Schematic KAT and Flowchart Schemes

To apply KAT in program development and verification, formal treatment of assignments and program states is required. Axioms for assignments have been added, for instance, in *schematic Kleene algebra with tests* (SKAT) [2]. This extension of KAT is targeted at modelling the transformation of flowchart schemes. A classical reference for flowchart schemes, scheme equivalence, and transformation is Manna's book *Mathematical Theory of Computation* [21]. Our formalisation of SKAT in Isabelle is discussed in this section; our formalisation of a complex flowchart equivalence proof [21,2] is presented in Section 5. We describe the conceptual development of SKAT together with its formalisation in Isabelle.

A *ranked alphabet* or signature Σ consists of a family of function symbols f, g, ... and relation symbols P, Q, ... together with an arity function mapping symbols to $\mathbb{N}$. There is always a null function symbol with arity 0. In Isabelle, we have implemented ranked alphabets as a type class. Variables are represented by natural numbers. Terms over Σ are defined as a polymorphic Isabelle datatype.

```
datatype 'a trm = App 'a "'a trm list" | Var nat
```

We omit arity checks to avoid polluting proofs with side conditions. In practice, verifications will fail if arities are violated. Variables and Σ-terms form assignment statements; together with predicate symbols they form tests in SKAT. Predicate expressions (atomic formulae) are also implemented as a datatype.

```
datatype 'a pred = Pred 'a "'a trm list"
```

Evaluation of terms, predicates and tests relies on an interpretation function. It maps function and relation symbols to functions and relations. It is used to define a notion of flowchart equivalence [2,21] with respect to all interpretations. It is also needed to formalise Hoare logic in Section 6 by interpreting Σ-expressions in semantic domains. In Isabelle, it is based on the following pair of functions.

```
record ('a, 'b) interp =
  interp-fun :: 'a ⇒ 'b list ⇒ 'b
  interp-rel :: 'a ⇒ 'b relation
```

We can now include Σ-expressions into SKAT expressions, which model flowchart schemes.

```
datatype 'a skat-expr =
    SKAssign nat "'a trm"
  | SKPlus "'a skat-expr" "'a skat-expr" (infixl "⊕" 70)
  | SKMult "'a skat-expr" "'a skat-expr" (infixl "⊙" 80)
  | SKStar "'a skat-expr"
  | SKBool "'a pred bexpr"
  | SKOne
  | SKZero
```

In this datatype, *SKAssign* is the assignment constructor; it takes a variable and a Σ-term as arguments. The other constructors capture the programming constructs of sequential composition, conditionals and while loops within KAT. The type *'a pred bexpr* represents Boolean combinations of predicates, which form the tests in SKAT. The connection between the SKAT syntax and Manna's flowchart schemes is discussed in [2], but we do not formalise it.

Having formalised the SKAT syntax we can now define a notion of flowchart equivalence by using Isabelle's quotient types. First we define the obvious congruence on SKAT terms that includes the KAT axioms and the SKAT assignment axioms

$$\begin{aligned}
x := s; y := t &= y := t[x/s]; x := s && (y \notin FV(s)),\\
x := s; y := t &= x := s; y := t[x/s] && (x \notin FV(s)),\\
x := s; x := t &= x := t[x/s],\\
a[x/t]; x := t &= x := t; a.
\end{aligned}$$

In the following inductive definition we only show the equivalence axioms, a single Kleene algebra axiom and an assignment axiom explicitly. Additional recursive functions for free variables and substitutions support the assignment axioms.

inductive *skat-cong* :: *('a::ranked-alphabet) skat-expr* $\Rightarrow$ *'a skat-expr* $\Rightarrow$ *bool*
(**infix** $\approx$ *55*) **where**
refl [*intro*]: $p \approx p$
| *sym* [*sym*]: $p \approx q \Longrightarrow q \approx p$
| *trans* [*trans*]: $p \approx q \Longrightarrow q \approx r \Longrightarrow p \approx r$
...
| *mult-assoc*: $(p \odot q) \odot r \approx p \odot (q \odot r)$
...
| *assign1*: $[\![x \neq y;\ y \notin FV\ s]\!] \Longrightarrow$
SKAssign x s $\odot$ *SKAssign y t* $\approx$ *SKAssign y* $(t[x/s])$ $\odot$ *SKAssign x s*
...

Isabelle's quotient package [17] now allows us to formally take the quotient of SKAT expressions with respect to *skat-cong*. The SKAT axioms then become available for reasoning about SKAT expressions.

quotient-type *'a skat* = *('a::ranked-alphabet) skat-expr / skat-cong*

Using this notion of equivalence on SKAT expressions we can define additional syntactic sugar by lifting constructors to SKAT operations, for instance,

lift-definition *skat-plus* :: *('a::ranked-alphabet) skat* $\Rightarrow$ *'a skat* $\Rightarrow$ *'a skat*
(**infixl** + *70*) **is** *SKPlus*

We have used Isabelle's transfer tactic to provide nice programming syntax and lift definitions from the congruence. For instance,

lemma *skat-assign1*:
$[\![x \neq y;\ y \notin FV\ s]\!] \Longrightarrow (x := s \cdot y := t) = (y := t[x/s] \cdot x := s)$

An interpretation statement formally shows in Isabelle that the algebra thus constructed forms a KAT.

definition *tests* :: *('a::ranked-alphabet) skat ord* **where**
$tests = (\!|carrier = test\text{-}set,\ le = (\lambda p\ q.\ skat\text{-}plus\ p\ q = q)|\!)$

definition *free-kat* :: *('a::ranked-alphabet) skat test-algebra* **where**
$free\text{-}kat = (\!|carrier = UNIV,\ plus = skat\text{-}plus,\ mult = skat\text{-}mult,\ one = skat\text{-}one,$
$zero = skat\text{-}zero,\ star = skat\text{-}star,\ test\text{-}algebra.test = tests|\!)$

interpretation *skt*: *kat free-kat*

Proving this statement required some work. First, is uses our comprehensive implementation of Kleene algebra with tests (and with carrier sets) in Isabelle. Second, we needed to show that the quotient algebra constructed satisfies the KAT axioms, including those of Boolean algebra for the subalgebra of tests. A main complication comes from the fact that Boolean complementation is defined as a partial operation, that is, on tests only; thus it cannot be directly lifted from the congruence. We have defined it indirectly using Isabelle's indefinite description operator. After this interpretation proof, most statements shown for KAT are automatically available in the quotient algebra. The unfortunate exception is again the partially defined negation symbol, which is not fully captured by the interpretation statement. Here, KAT theorems need to be duplicated by hand.

When defining a quotient type, Isabelle automatically generates two coercion functions. The *abs-skat* function maps elements of type *'a skat-expr* to elements of the quotient algebra type *'a skat*, while the *rep-skat* function maps in the converse direction. Both these functions are again based on Isabelle's definite description operator, which can be unwieldy. However, as our types are inductively defined, we can as well use the following equivalent, and computationally more appealing, recursive function instead of *abs-skat*, which supports simple proofs by induction.

primrec *abs* :: *('a::ranked-alphabet) skat-expr* ⇒ *'a skat* (⌊-⌋ [*111*] *110*) **where**
$abs\ (SKAssign\ x\ t) = x := t$
$|\ abs\ (SKPlus\ p\ q) = abs\ p + abs\ q$
$|\ abs\ (SKMult\ p\ q) = abs\ p \cdot abs\ q$
$|\ abs\ (SKBool\ a) = test\text{-}abs\ a$
$|\ abs\ SKOne = \mathbf{1}$
$|\ abs\ SKZero = \mathbf{0}$
$|\ abs\ (SKStar\ p) = (abs\ p)^\star$

Mathematically, *abs* (or $\lfloor - \rfloor$) is a homomorphism. It is useful for programming various tactics.

4 Formalising a Metatheorem

We have formalised a metatheorem due to Angus and Kozen (Lemma 4.4 in [2]) that can be instantiated, for instance, to check commutativity conditions, eliminate redundant variables or rename variables in flowchart transformation proofs. We instantiate this theorem mainly to develop tactics that support proof automation in the flowchart example of the next section.

theorem *metatheorem*:
assumes *kat-homomorphism f*
and *kat-homomorphism g*
and $\bigwedge a.\ a \in atoms\ p \Longrightarrow f\ a \cdot q = q \cdot g\ a$
shows $f\ p \cdot q = q \cdot g\ p$

We proceed by induction on p, expanding Angus and Kozen's proof. The predicate *kat-homomorphism* in the theorem states that f and g are KAT morphisms. This notion is defined in Isabelle as a locale in the obvious way. The functions f and g map from SKAT terms into the SKAT quotient algebra, hence they have the same type as *abs*. The atoms function returns all the atomic subexpressions of a SKAT term, i.e. all the assignments and atomic tests.

Angus and Kozen have observed that if q commutes with all atomic subexpressions of p, then q commutes with p. This is a simple instantiation of the metatheorem. It can be obtained in Isabelle as follows:

lemmas *skat-comm* = *metatheorem*[*OF abs-hom abs-hom*]

This instantiates f and g using the fact that *abs* is a KAT morphism.

Lemma 4.5 in [2] states that if a variable x is not read in an expression p, then setting it to null will eliminate it from p.

lemma *eliminate-variables*:
assumes $x \notin reads\ p$
shows $\lfloor p \rfloor \cdot x := null = \lfloor eliminate\ x\ p \rfloor \cdot x := null$

In the statement of this lemma, *reads p* is a recursive function that returns all the variables on the right-hand side of all assignments within p, and the function *eliminate x p* removes all assignments to x in p.

We have used the metatheorem and its instances to develop tactics that check for commutativity and eliminate variables. These tactics take expressions of the quotient algebra and coerce them into the term algebra to perform these syntactic manipulations. All the machinery for these coercions, such as *abs*, is thereby hidden from the user. A simple application example is given by the following lemma.

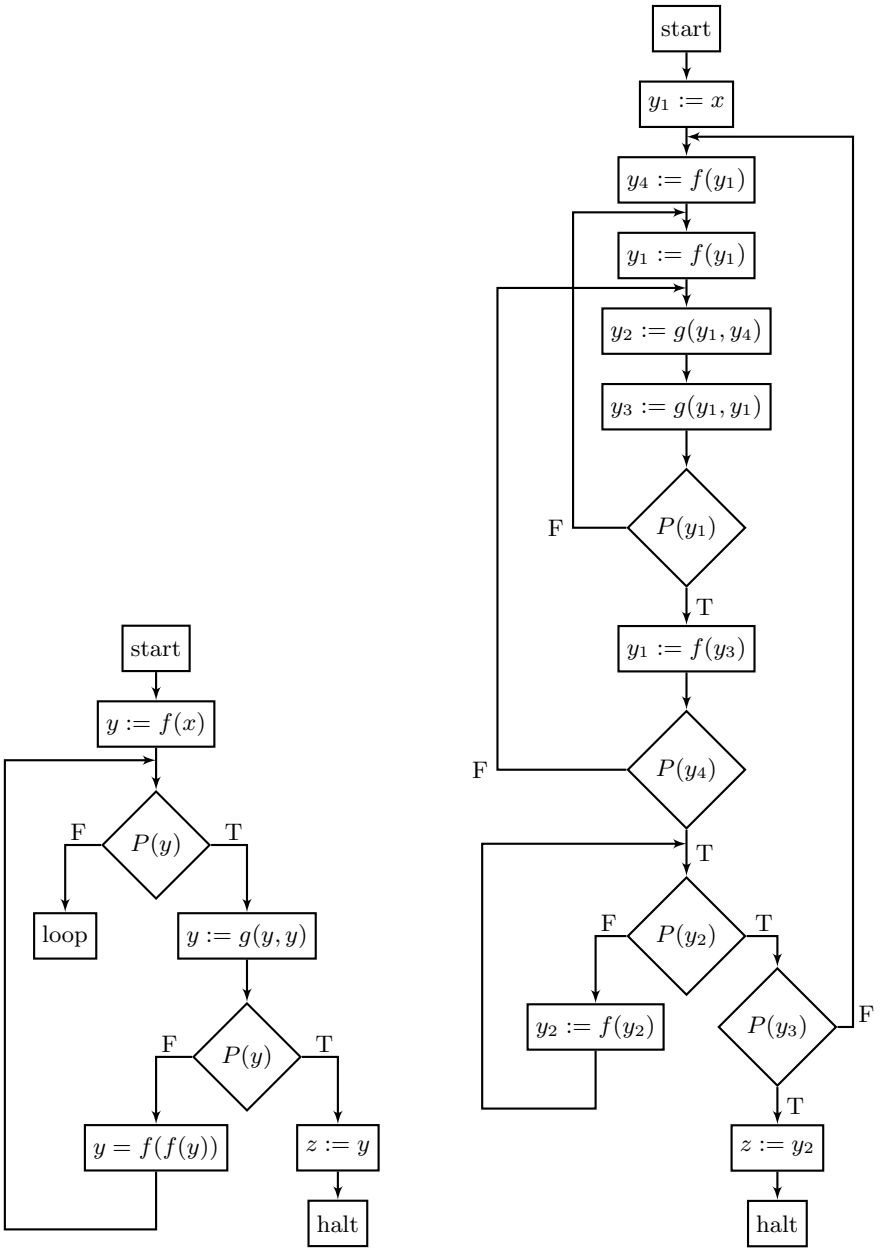
lemma *comm-ex: (1 := Var 2; 3 := Var 4) = (3 := Var 4; 1 := Var 2)*
by *skat-comm*

5 Verification of Flowchart Equivalence

We have applied our SKAT implementation to verify a well known flowchart equivalence example in Isabelle. It is attributed by Manna to Paterson [21]. The flowcharts can be found at page 16f. in Angus and Kozen's paper [2] or pages 254 and 258 in Manna's book [21]; they are reproduced here in Figure 1. Manna's proof essentially uses diagrammatic reasoning, whereas Angus and Kozen's proof is equational. We reconstruct the algebraic proof at the same level of granularity in Isabelle. The two flowcharts, translated into SKAT by Angus and Kozen, are as follows.

```
definition scheme1 ≡ seq
 [ 1 := vx, 4 := f (Var 1), 1 := f (Var 1)
 , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
 , loop
   [ !(P (Var 1)), 1 := f (Var 1)
   , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
   ]
 , P (Var 1), 1 := f (Var 3)
 , loop
   [ !(P (Var 4)) + seq
     [ P (Var 4)
     , (!(P (Var 2)); 2 := f (Var 2))⋆
     , P (Var 2), ! (P (Var 3))
     , 4 := f (Var 1), 1 := f (Var 1)
     ]
   , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
   , loop
     [ !(P (Var 1)), 1 := f (Var 1)
     , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
     ]
   , P (Var 1), 1 := f (Var 3)
   ]
 , P (Var 4)
 , (!(P (Var 2)) · 2 := f (Var 2))⋆
 , P (Var 2), P (Var 3), 0 := Var 2, halt
 ]

definition scheme2 ≡ seq
 [ 2 := f vx, P (Var 2)
 , 2 := g (Var 2) (Var 2)
 , loop
   [ !(P (Var 2))
   , 2 := f (f (Var 2))
   , P (Var 2)
   , 2 := g (Var 2) (Var 2)
   ]
 , P (Var 2), 0 := Var 2, halt
 ]
```



Scheme S_{6E} [21, p. 258] Scheme S_{6A} [21, p. 254]

Fig. 1. Two equivalent flowchart schemes

In the code, lists delimited by brackets indicate blocks of sequential code; loop expressions indicate the star of a block of code that follows. The *seq* function converts a block of code into a SKAT expression. The *halt* command sets all non output variables used in the scheme to *null*. To make algebraic reasoning more efficient, we follow Angus and Kozen in introducing definitions that abbreviate atomic commands, in particular assignments, and tests. The flowchart equivalence problem can then be expressed more succinctly and abstractly in KAT, as all assignment statements, which are dealt with by SKAT, have been abstracted.

$$
\begin{aligned}
&\mathit{seq}\ [x1,p41,p11,q214,q311,\mathit{loop}\ [!a1,p11,q214,q311],a1,p13\\
&\quad ,\mathit{loop}\ [!a4\ +\ \mathit{seq}\ [a4,(!a2\cdot p22)^\star,a2,!a3,p41,p11]\\
&\quad ,q214,q311,\mathit{loop}\ [!a1,p11,q214,q311],a1,p13]\\
&\quad ,a4,(!a2\cdot p22)^\star,a2,a3,z2,\mathit{halt}]\\
=\\
&\mathit{seq}\ [s2,a2,q222,(\mathit{seq}\ [!a2,r22,a2,q222])^\star,a2,z2,\mathit{halt}]
\end{aligned}
$$

The proof that rewrites these KAT expressions, however, needs to descend to SKAT in order to derive commutativity conditions between expressions that depend on variables and Σ-terms. These conditions are then lifted to KAT. The condition expressed in Lemma *comm-ex* from Section 4, for instance, reduces to the KAT identity $pq = qp$ when abbreviating $1 := Var\ 2$ as p, and $3 := Var\ 4$ as q. In our proof we infer these commutativity conditions in a lazy fashion. This follows Angus and Kozen's proof essentially line by line.

We heavily depend on our underlying KAT library, which contains about 100 lemmas for dealing with the Kleene star and combined reasoning about the interaction between actions and tests. Typical properties are $(p + q)^* = p^*(q \cdot p^*)^*$, $(pq)^*p = p(qp)^*$ or $bp = pc \Longleftrightarrow bp!c = !bpc$. We have also refined the tactics mentioned in the previous section to be able to efficiently manipulate the large SKAT expressions that occur in the proof. Most of these implement commutations in lists of expressions modulo commutativity conditions on atomic expressions which are inferred from SKAT terms on the fly.

The size of our proof as a LaTeX document is about 12 pages, twice as many as in Angus and Kozen's manual proof, but this is essentially due to aligning their horizontal equational proofs in a vertical way. A previous proof in a special-purpose SKAT prover required 41 pages [1]. This impressively demonstrates the power of Isabelle's proof automation. Previous experience in theorem proving with algebra shows that the level of proof automation in algebra is often very high [15,13,12]. In this regard, our present proof experience is slightly underwhelming, as custom tactics and low-level proof techniques were needed for our step-by-step proof reconstruction. A higher degree of automation seems difficult to achieve, and a complete automation of the scheme equivalence proof currently out of reach. The main reason is that the flowchart terms in KAT are much longer, and combinatorially more complex, than those in typical textbook proofs. Decision procedures for variants of Kleene algebras, which currently only exist in Coq [6], might overcome this difficulty.

6 Hoare Logic

It is well known that Hoare logic—except the assignment rule—can be encoded in KAT as well as in other variants of Kleene algebra such as modal Kleene algebras [22] and Kleene modules [10]. The latter are algebraic relatives of propositional dynamic logic. A combination of these algebras with the assignment rule and their application in program verification has so far not been attempted.

We have implemented a novel approach in which SKAT and Kleene modules are combined. This allows us to separate tests conceptually from the pre- and post-conditions of programs.

A *Kleene module* [20] is a structure $(K, L, :)$ where K is a Kleene algebra, L a join-semilattice with least element $\bot$ and : a mapping of type $L \times K \to L$ where

$$\begin{aligned} P : (p + q) &= P : p \sqcup P : q, & (P \sqcup Q) : p &= P : p \sqcup Q : p, \\ P : (p \cdot q) &= P : p : q, & (P \sqcup Q) : p \leq Q &\longrightarrow P : p^* \leq Q, \\ P : 0 &= \bot, & P : 1 &= P. \end{aligned}$$

In this context, L models the space of states, propositions or assertions of a program, K its actions, and the scalar product maps a proposition and an action to a new proposition. We henceforth assume that L is a Boolean algebra with maximal element $\top$ and use a KAT instead of a Kleene algebra as the first component of the module. The interaction between assertions, as modelled by the Boolean algebra L, and tests, as modelled by the Boolean algebra B, is captured by the new axiom

$$P : a = P \sqcap (\top : a).$$

The scalar product $\top : a$ coerces the test a into an assertion ($\top$ does not restrict it); the scalar product $P : a$ is therefore equal to a conjunction between the assertion P and the test a.

We have used Isabelle's locales to implement modules over KAT. Hoare triples can then be defined as usual.

definition *hoare-triple* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow$ *bool* ($\{\!|$-$|\!\}$ - $\{\!|$-$|\!\}$ [*54*,*54*,*54*] *53*) **where**
$\{\!|P|\!\}\ p\ \{\!|Q|\!\} \equiv P :: p \sqsubseteq_L Q$

As : is a reserved symbol in Isabelle, we use :: for the scalar product. The index L refers to the Boolean algebra of assertions and the order $\sqsubseteq_L$ is the order on this Boolean algebra. As is well known, the Hoare rules excluding assignment can now be derived as theorems in these modules more or less automatically. Applying the resulting Hoare-style calculus—which is purely equational—for program verification requires us to provide more fine-grained syntax for assertions and refinement statements and adding some form of assignment axiom.

We obtain this first-order syntax once more by specialising KAT to SKAT, and by interpreting the SKAT expressions in the Boolean algebra of propositions or states. As usual, program states are represented as functions from variables to values. Assertions correspond to sets of states. Hence the Boolean algebra L

is instantiated as a powerset algebra over states. Similar implementations are already available in theorem provers such as Isabelle, HOL and Coq [24,26,8,23], but they have not been implemented as simple instantiations of more general algebraic structures. Assignment statements are translated in Gordon style [8] into forward predicate transformers which map assertions (preconditions) to assertions (postconditions).

This is, of course, compatible with the module-based approach. To implement the scalar product of our KAT module, we begin by writing an evaluation function which, given an interpretation and a SKAT expression, returns the forward predicate transformer for that expression.

fun *eval-skat-expr* ::
  (*'a*::*ranked-alphabet*, *'b*) *interp* ⇒ *'a skat-expr* ⇒ *'b mems* ⇒ *'b mems*
  where
  eval-skat-expr D (*SKAssign x t*) *Δ* = *assigns D x t Δ*
| *eval-skat-expr D* (*SKBool a*) *Δ* = *filter-set* (*eval-bexpr D a*) *Δ*
| *eval-skat-expr D* (*p* ⊙ *q*) *Δ* = *eval-skat-expr D q* (*eval-skat-expr D p Δ*)
| *eval-skat-expr D* (*p* ⊕ *q*) *Δ* = *eval-skat-expr D p Δ* ∪ *eval-skat-expr D q Δ*
| *eval-skat-expr D* (*SKStar p*) *Δ* = (⋃ *n*. *iter n* (*eval-skat-expr D p*) *Δ*)
| *eval-skat-expr D SKOne Δ* = *Δ*
| *eval-skat-expr D SKZero Δ* = {}

We can now prove that if two SKAT expressions are equivalent according to the congruence defined in Section 3, then they represent the same predicate transformer. The proof is by induction. This property allows us to lift the *eval-skat-expr* function to the quotient algebra.

theorem *skat-cong-eval*:
  skat-cong p q ⟹ ∀ *Δ*. *eval-skat-expr D p Δ* = *eval-skat-expr D q Δ*

lift-definition *eval* ::
  (*'a*::*ranked-alphabet*, *'b*) *interp* ⇒ *'a skat* ⇒ *'b mems* ⇒ *'b mems*
  is *eval-skat-expr*

Using this lifting, we can reason algebraically in instances of SKAT that have been generated by the evaluation function. This enables us to derive an assignment rule for forward reasoning in Hoare logic from the SKAT axioms.

lemma *hoare-assignment*: $P[x/t] \subseteq Q \Longrightarrow \{\!|P|\!\}\ x := t\ \{\!|Q|\!\}$

We could equally derive a forward assignment rule $P\{\!|x := s|\!\}P[x/s]$, but this seems less useful in practice.

To facilitate automated reasoning we have added a notion of loop invariant as syntactic sugar for while loops. Invariants are assertions used by the tactic that generates verification conditions.

$$\mathit{WHILE}\ b\ \mathit{INVARIANT}\ i\ \mathit{DO}\ p\ \mathit{WEND} = (bp)^*\overline{b}.$$

We have also derived a refined while rule which uses the loop invariant.

lemma *hoare-while-inv*:
assumes *b-test*: $b \in carrier\ tests$
and *Pi*: $P \subseteq i$ **and** *iQ*: $i \cap (UNIV :: !b) \subseteq Q$
and *inv-loop*: $\{\!|i \cap (UNIV :: b)|\!\}\ p\ \{\!|i|\!\}$
shows $\{\!|P|\!\}\ WHILE\ b\ INVARIANT\ i\ DO\ p\ WEND\ \{\!|Q|\!\}$

This particular rule has been instantiated to the powerset algebra of states, but it could as well have been defined abstractly.

Isabelle already provides a package for Hoare logic [26]. Since there is one Hoare rule per programming construct, it uses a tactic to blast away the control structure of programs. We have implemented a similar tactic for our SKAT-based implementation, called *hoare-auto*.

7 Verification Examples

We have applied our variant of Hoare logic to prove the partial correctness of some simple algorithms. Instead of applying each rule manually we use our tactic *hoare-auto* to make their verification with sledgehammer almost fully automatic. More complex examples would certainly require more user interaction or more sophisticated tactics to discharge the generated proof obligations.

lemma *euclids-algorithm*:
$\{\!|\{mem.\ mem\ 0 = x \wedge mem\ 1 = y\}|\!\}$
$WHILE\ !(pred\ (EQ\ (Var\ 1)\ (NAT\ 0)))$
$INVARIANT\ \{mem.\ gcd\ (mem\ 0)\ (mem\ 1) = gcd\ x\ y\}$
DO
$2 := Var\ 1;$
$1 := MOD\ (Var\ 0)\ (Var\ 1);$
$0 := Var\ 2$
$WEND$
$\{\!|\{mem.\ mem\ 0 = gcd\ x\ y\}|\!\}$
by *hoare-auto* (*metis gcd-red-nat*)

lemma *factorial*:
$\{\!|\{mem.\ mem\ 0 = x\}|\!\}$
$1 := NAT\ 1;$
$(WHILE\ !(pred\ (EQ\ (Var\ 0)\ (NAT\ 0)))$
$INVARIANT\ \{mem.\ fact\ x = mem\ 1 * fact\ (mem\ 0)\}$
DO
$1 := MULT\ (Var\ 1)\ (Var\ 0);\ 0 := MINUS\ (Var\ 0)\ (NAT\ 1)$
$WEND)$
$\{\!|\{mem.\ mem\ 1 = fact\ x\}|\!\}$
by *hoare-auto* (*metis fact-reduce-nat*)

Finally, our algebraic approach is expressive enough for deriving further program transformation or refinement rules, which would only be admissible in Hoare logic. As an example we provide proofs of two simple Hoare-style inference rules. Program refinement or transformation rules could be derived in a similar way.

lemma *derived-rule1*:
assumes $\{P1,P2,Q1,Q2\} \subseteq$ *carrier A* **and** $p \in$ *carrier K*
and $\{\!|P1|\!\}\ p\ \{\!|Q1|\!\}$ **and** $\{\!|P2|\!\}\ p\ \{\!|Q2|\!\}$
shows $\{\!|P1 \sqcap P2|\!\}\ p\ \{\!|Q1 \sqcap Q2|\!\}$
using *assms*
apply (*auto simp add*: *hoare-triple-def assms*, *subst A.bin-glb-var*)
by (*metis A.absorb1 A.bin-lub-var A.meet-closed A.meet-comm mod-closed mod-join*)+

lemma *derived-rule2*:
assumes $\{P,Q,R\} \subseteq$ *carrier A* **and** $p \in$ *carrier K* **and** $P :: p = (\top :: p) \sqcap P$
and $\{\!|Q|\!\}\ p\ \{\!|R|\!\}$
shows $\{\!|P \sqcap Q|\!\}\ p\ \{\!|P \sqcap R|\!\}$
by (*insert assms*) (*smt derived-rule1 derived-rule2 insert-subset*)

Only the derivation of the first rule is not fully automatic. The side condition $P :: p = (\top :: p) \sqcap P$ expresses the fact that if assertion P holds before execution of program p, which is the left-hand side of the equation, then it also holds after p is executed. The expression $\top :: p$ represents the assertion that holds after p is executed without any input restriction.

These examples demonstrate the benefits of the algebraic approach in defining syntax, deriving domain-specific inference rules and linking with more refined models and semantics of programs with exceptional ease. While, in the context of verification, these tasks belong more or less to the metalevel, they are part of actual correctness proofs in program construction, transformation or refinement. We believe that this will be the most important domain for future applications.

8 Conclusion

We have implemented schematic Kleene algebra with tests in Isabelle/HOL, and used it to formalise a complex flowchart equivalence proof by Angus and Kozen. Our proof is significantly shorter than a previous formalisation in a custom theorem prover for Kleene algebra with tests. Our proof follows Angus and Kozen's manual proof almost exactly and translates it essentially line-by-line into Isabelle, despite some weaknesses in proof automation which sometimes forced us to reason at quite a low level. We have also extended SKAT to support the verification of simple algorithms in a Hoare-logic style. Our approach provides a seamless bridge between our abstract algebraic structures and concrete programs. We have tested our approach on a few simple verification examples. Beyond that, we have derived additional Hoare-style rules and tactics for proof automation abstractly in the algebraic setting. These can be instantiated to different semantics and application domains. In the context of verification the main aplications of algebra seem to be at this meta-level. The situation is different when developing programs from specifications or proving program equivalence, as the flowchart scheme tranformation shows. In this context, algebra can play an essential role in concrete proofs.

Our Isabelle implementation sheds some light on the role of Kleene algebra for program development and verification. Given libraries for the basic algebraic structures and important models, we could prototype tools for flowchart equivalence and Hoare-style verification proofs with little effort and great flexibility. Moving, e.g., from a partial to a total correctness environment would require minor changes to the algebra (and of course a termination checker). We doubt that a bottom-up semantic approach would be equally simple and flexible. Isabelle turned out to be very well suited for our study. Hierarchies of algebras and their models could be implemented using type classes and locales, verification tasks were well supported by tactics and automated theorem provers. The automation of textbook algebraic proofs is usually very high. Algebraic proof obligations generated from verification conditions, however, turned out to be more complex, cf. our flowchart example. Such proofs can provide valuable benchmarks for developers of theorem provers, decision procedures and domain specific solvers.

References

1. Aboul-Hosn, K., Kozen, D.: KAT-ML: An interactive theorem prover for Kleene algebra with tests. Journal of Applied Non-Classical Logics 16(1-2), 9–34 (2006)
2. Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University (July 2001)
3. Armstrong, A., Struth, G.: Automated reasoning in higher-order regular algebra. In: Kahl, W., Griffin, T.G. (eds.) RAMICS 2012. LNCS, vol. 7560, pp. 66–81. Springer, Heidelberg (2012)
4. Armstrong, A., Struth, G., Weber, T.: Kleene algebra. Archive of Formal Proofs, Formal proof development (2013), `http://afp.sf.net/entries/Kleene_Algebra.shtml`
5. Berghammer, R., Struth, G.: On automated program construction and verification. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 22–41. Springer, Heidelberg (2010)
6. Braibant, T., Pous, D.: Deciding Kleene algebras in Coq. Logical Methods in Computer Science 8(1) (2012)
7. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
8. Collavizza, H., Gordon, M.: Forward with Hoare. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C. A. R. Hoare, pp. 101–121. Springer, Heidelberg (2010)
9. Desharnais, J., Möller, B., Struth, G.: Algebraic notions of termination. Logical Methods in Computer Science 7(1) (2011)
10. Ehm, T., Möller, B., Struth, G.: Kleene modules. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 112–124. Springer, Heidelberg (2004)
11. Fernandes, T., Desharnais, J.: Describing data flow analysis techniques with Kleene algebra. Sci. Comput. Program. 65(2), 173–194 (2007)
12. Foster, S., Struth, G.: Automated analysis of regular algebra. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 271–285. Springer, Heidelberg (2012)

13. Guttmann, W., Struth, G., Weber, T.: Automating algebraic methods in Isabelle. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 617–632. Springer, Heidelberg (2011)
14. Guttmann, W., Struth, G., Weber, T.: A repository for Tarski-Kleene algebras. In: Höfner, P., McIver, A., Struth, G. (eds.) ATE 2011. CEUR Workshop Proceedings, vol. 760, pp. 30–39. CEUR-WS.org (2011)
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. Huffman, B., Kunčar, O.: Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In: Isabelle Users Workshop (2012)
17. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C.-C. (eds.) SAC, pp. 1639–1644. ACM (2011)
18. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. 19(3), 427–443 (1997)
19. Kozen, D., Patron, M.-C.: Certification of compiler optimizations using Kleene algebra with tests. In: Palamidessi, C., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 568–582. Springer, Heidelberg (2000)
20. Leiß, H.: Kleene modules and linear languages. J. Log. Algebr. Program. 66(2), 185–194 (2006)
21. Manna, Z.: Mathematical theory of computation. McGraw-Hill (1974)
22. Möller, B., Struth, G.: Algebras of modal operators and partial correctness. Theor. Comput. Sci. 351(2), 221–239 (2006)
23. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Dependent types for imperative programs. In: Hook, J., Thiemann, P. (eds.) ICFP, pp. 229–240. ACM (2008)
24. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics. Formal Asp. Comput. 10(2), 171–186 (1998)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
26. Schirmer, N.: Verification of sequential imperative programs in Isabelle-HOL. PhD thesis, Technische Universität München (2006)

Pragmatic Quotient Types in Coq

Cyril Cohen

Department of Computer Science and Engineering
University of Gothenburg
cyril.cohen@gu.se

Abstract. In intensional type theory, it is not always possible to form the quotient of a type by an equivalence relation. However, quotients are extremely useful when formalizing mathematics, especially in algebra. We provide a Coq library with a pragmatic approach in two complementary components. First, we provide a framework to work with quotient types in an axiomatic manner. Second, we program construction mechanisms for some specific cases where it is possible to build a quotient type. This library was helpful in implementing the types of rational fractions, multivariate polynomials, field extensions and real algebraic numbers.

Keywords: Quotient types, Formalization of mathematics, Coq.

Introduction

In set-based mathematics, given some base set S and an equivalence $\equiv$, one may see the quotient $(S \,/\, \equiv)$ as the partition $\{\pi(x) \mid x \in S\}$ of S into the sets $\pi(x) \mathrel{\hat{=}} \{y \in S \mid x \equiv y\}$, which are called equivalence classes.

We distinguish several uses of quotients in the literature. On the one hand, we have structuring quotients, where the quotient can often be equipped with more structure than the base set. For instance, the quotient of pairs of integers to get rational numbers can be equipped with a field structure. Similarly, a quotient of the free algebra of terms generated by constants, variables, sums and products gives multivariate polynomials (*i.e.* polynomials with arbitrarily many variables). This kind of quotient is often left implicit in mathematical papers.

On the other hand, we have algebraic quotients, for which we can transfer the structure from the base set to the quotient. For instance, the quotient of a group by a normal subgroup or the quotient of a ring by an ideal belong to this category. For this kind of quotient, the structure on the base set and on the quotient set matter and the canonical surjection onto the quotient is a morphism for this structure.

In type theory, there are two known options to represent the notion of quotient. The first option is to consider quotients of setoids. A setoid is a type with an equivalence relation called setoid equality [1]. Now, quotienting a setoid amounts to changing the setoid equality to a broader one. However, we still consider elements from the base type, *i.e.* the type underlying both the base setoid and the quotient setoid. This point of view is more the study of equivalence

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 213–228, 2013.

relations than the study of quotients. Moreover, although rewriting with setoid equality is supported by the system [2], it is still not as practical nor efficient as rewriting with Leibniz equality, because it must check the context (of the term to rewrite) commutes with the setoid equality.

The second option, and the one we focus on is to forge a quotient type, *i.e.* a type where each element represents one and only one equivalence class of the base type. This point of view leads to study the quotient as a new type, on which equality is the standard (Leibniz) equality of the system. In this framework, the equivalence that led to the quotient type is not a primitive notion. In Coq, the problem with quotient types is that there is no general way of forming them, without axioms [3].

In this paper, we do not focus on the theoretical problem of existence of quotients, but on the way to use them. We first describe our definition of a quotient interface in Section 1. We also show in which way it captures the desired properties of quotients and how we instrumented type inference to help the user to be concise. Surprisingly, this interface does not rely on an equivalence relation in its axiomatization, so we explain how we recover quotients by an equivalence relation from it in Section 2. We provide in Section 3 examples of applications of quotient types to rational fractions, multivariate polynomials, field extensions and real algebraic numbers. In Section 4, we compare our interface to previous designs of quotient types.

The Coq code for this framework and the examples are available at the following address: `http://perso.crans.org/cohen/work/quotients/`. We use the SSReflect tactic language [4] and the mathematical components project libraries [5].

1 A Framework for Quotients

Definition 1 (Quotient type). *A type* `Q` *is a* quotient type *of a base type* `T` *if there are two functions* (`pi: T -> Q`) *and* (`repr: Q -> T`) *and a proof* `reprK`[1] *that* `pi` *is a left inverse for* `repr`*, i.e.* `forall x, pi (repr x)= x` *(see Figure 1).*

The function `pi` *is called the* canonical surjection *and we call the function* `repr` *the* representative function.

1.1 A Small Interface

The interface for quotients has a field for the quotient type, a field for the representative function `repr`, a field for the canonical surjection `pi` and a field for the axiom `reprK`, which says the representative function is a section of the canonical surjection. The definition of the quotient interface is split into two parts, in the same way the interfaces from the SSReflect algebraic hierarchy are [6], in order to improve modularity.

[1] The name `reprK` comes from a standard convention in the SSReflect library to use the suffix "K" for cancellation lemmas.

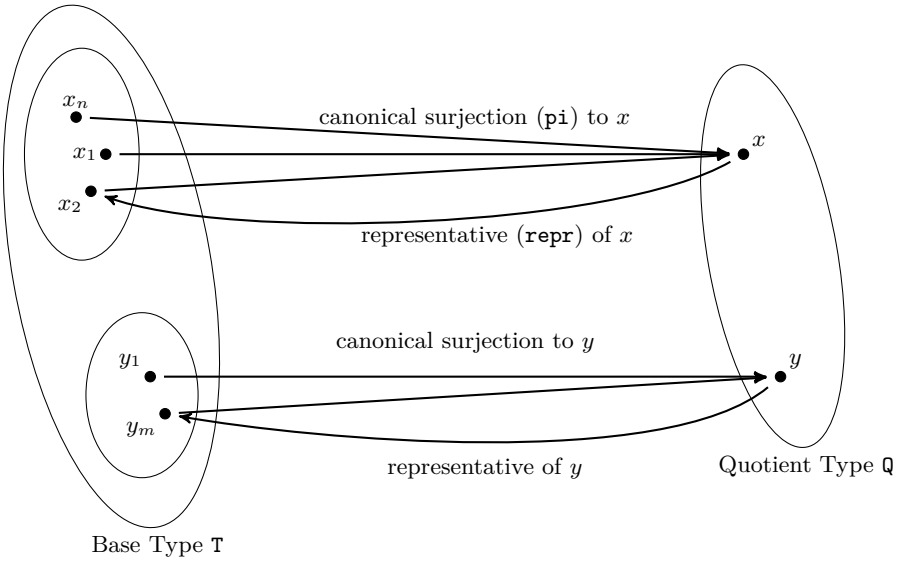


Fig. 1. Quotients without equivalence relation

```
Record quot_class_of (T Q : Type) := QuotClass {
  repr : Q -> T;
  pi : T -> Q;
  reprK : forall x : Q, pi (repr x) = x
}.
Record quotType (T : Type) := QuotType {
  quot_sort :> Type; (* quotient type *)
  quot_class : quot_class_of T quot_sort (* quotient class *)
}
```

Definition 2 (Quotient structure). *An instance of the quotient interface is called a* quotient structure.

Example 1. Let us define the datatype `int` of integers as the quotient of pairs of natural numbers by the diagonal. In other words, integers are the quotient of $\mathbb{N} \times \mathbb{N}$ by the equivalence relation

$$((n_1, n_2) \equiv (m_1, m_2)) \quad \widehat{=} \quad (n_1 + m_2 = m_1 + n_2).$$

Now, we explicitly define the type of canonical representatives: pairs of natural numbers such that one of them is zero. For example, the integer zero is represented by $(0, 0)$, one by $(1, 0)$ and minus one by $(0, 1)$.

```
Definition int_axiom (z : nat * nat) := (z.1 == 0) || (z.2 == 0).
Definition int := {z | int_axiom z}.
```

We then define the particular instances `reprZ` of `repr` and `piZ` of `pi` and we show that `reprZ` is indeed a section of `piZ`:

```
Definition reprZ (n : int) : nat * nat := val n.

Lemma sub_int_axiom x y : int_axiom (x - y, y - x).

Definition piZ (x : nat * nat) : int :=
  exist _ (x.1 - x.2, x.2 - x.1) (sub_int_axiom _ _).

Lemma reprZK x : piZ (reprZ x) = x.
```

where `val` is the first projection of a Σ-type and where `exist` is the constructor of a Σ-type.

Now, we pack together `reprZ`, `piZ` and `reprZK` into the quotient class, and in the quotient structure `int_quotType`.

```
Definition int_quotClass := QuotClass reprZK.
Definition int_quotType := QuotType int int_quotClass.
```

We created a data type `int` which is the candidate for being a quotient, and a structure `int_quotType` which packs `int` together with the evidence that it is a quotient.

Remark 1. For various reasons we do not detail here, we selected none of the implementation of `int` from this paper to define the integer datatype `int` from SSReflect library [7]. Our examples use `int` only for the sake of simplicity and we provide a compilation of the ones about `int` in the file `quotint.v`. However, this framework is used in practice for more complicated instances (see Section 3).

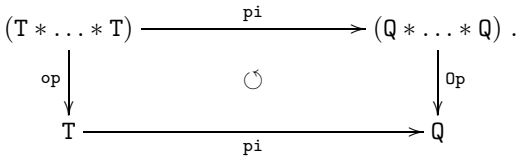
1.2 Recovering an Equivalence and Lifting Properties

The existence of a quotient type `Q` with its quotient structure `qT` over the base type `T` induces naturally an equivalence over `T`: two elements `(x y : T)` are equivalent if `(pi qT x = pi qT y)`. We mimic the notation of the SSReflect library for equivalence in modular arithmetic on natural numbers, which gives us the following notation for equivalence of `x` and `y` of type `T` modulo the quotient structure `qT`:

```
Notation "x = y %[mod qT]" := (pi qT x = pi qT y).
```

We say an operator on `T` (*i.e.* a function which takes its arguments in `T` and outputs an element of `T`) is compatible with the quotient if given two lists of arguments which are pairwise equivalent, the two outputs are also equivalent. In other words, an operator is compatible with the quotient if it is constant on each equivalence class, up to equivalence.

When an operator `op` on `T` is compatible with the quotient, it has a *lifting*, which means there exists an operator `Op` on the quotient type `Q` such that following diagram commutes:



The canonical surjection is a morphism for this operator.

For example, a binary operator `(Op : Q -> Q -> Q)` is a lifting for the binary operator `(op : T -> T -> T)` with regard to the quotient structure `qT` as soon as the canonical surjection `(pi qT : T -> Q)` is a morphism for this operator:

```
forall x y, pi qT (op x y) = Op (pi qT x) (pi qT y) :> Q
```

which can be re-expressed in a standardized form for morphisms of binary operators in SSReflect:

```
{morph (pi qT) : x y / op x y >-> Op x y}
```

Example 2. Let us define the add operation on `int` as the lifting of the point-wise addition on pairs of natural numbers.

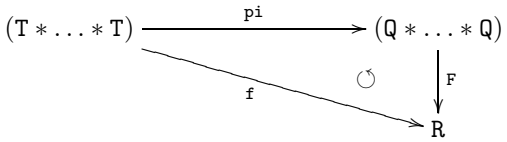
```
Definition add x y := (x.1 + y.1, x.2 + y.2).
Definition addz X Y := \pi_int (add (repr X) (repr Y)).
Lemma addz_compat : {morph \pi_int : x y / add x y >-> addz x y}.
```

Where the statement of `addz_compat` can be read as follows:

```
forall x y, \pi_int (add x y) = addz (\pi_int x) (\pi_int y).
```

and where `\pi_int` stands for `(pi int_quotType)` (see Example 4).

Similarly, given an arbitrary type `R`, we say that a function with values in `R` is compatible with the quotient if it is constant on each equivalence class. When a function `f` with arguments in `T` and values in `R` is compatible with the quotient, it has a *lifting*, which means there exists an operator `F` with arguments in the quotient type `Q` and values in `R` such that the following diagram commutes:



The canonical surjection is a morphism for this function.

For example, a binary function `(F : Q -> Q -> R)` is a lifting of a binary function `(f : T -> T -> R)` if:

```
forall x y, (f x y) = F (pi qT x) (pi qT y) :> R
```

or in a standardized form for morphisms of relations in SSReflect:

```
{mono (pi qT) : x y / f x y >-> F x y}
```

1.3 Inference of Quotient Structures

We recall that given a base type `T`, we say `Q` is a quotient type for `T` if there is a quotient structure `qT` and if the projection `(quot_sort qT)` is `Q`. In practice, given `x` in `Q` we want to be able to write `(repr x)`, but such a statement would be ill-typed.

Remark 2. The problem comes from the fact `repr` has an implicit argument which must have type `quotType`. The expanded form for `(repr x)` is `(@repr` $?_{\texttt{quotType}}$ `x)`, where `x` must have type `(quot_sort` $?_{\texttt{quotType}}$`)`. But if `x` has type `Q`, the type inference algorithm encounters the unification problem

$$(\texttt{quot\_sort}\ ?_{\texttt{quotType}}) \quad \equiv \quad \texttt{Q}$$

which it cannot solve without a hint, although we guess a solution is `qT`.

However, it is possible to make Coq type this statement anyway, by providing the information that the quotient structure `qT` is a *canonical structure* [8] for the quotient type `Q`. Registering a structure as canonical provide the unification mechanism a solution for the kind of equations we encounter.

Example 3. We make `int_quotType` the canonical quotient structure for the quotient type `int` by using the following Coq vernacular command:

```
Canonical Structure int_quotType.
```

Now, given `x` of type `int`, the system typechecks `(repr x)` as an element of `(nat * nat)`, as expected.

Since a quotient structure is canonically attached to every quotient type, we may also simplify the use of `pi`. Indeed, for now, `pi` has the following type.

```
forall (T : Type) (qT : quotType T), T -> quot_sort qT
```

Hence, `(pi qT)` has type `T -> Q`, but it is not possible to use `(pi Q)` to refer to this function. To circumvent this problem we provide a notation `\pi_Q` which gives exactly `(pi qT)` where `qT` is the canonical quotient structure attached to `Q` if it exists (otherwise, the notation fails). This notation uses a phantom type to let the system infer `qT` automatically [7].

Example 4. The canonical surjection function `\pi_int` from pairs of naturals to integers is in fact `(@pi int_quotType)` and has type `(nat * nat -> int)`.

We also adapted the notation for equivalence modulo quotient, so that we can provide `Q` instead of `qT`, as follows:

```
Notation "x = y %[mod Q]" := (\pi_Q x = \pi_Q y).
```

1.4 Automatic Rewriting

When an operation (or a function) is compatible with the quotient, we can forge the lifting by hand by composing the initial operation with `pi` and `repr`. In this case the canonical surjection is indeed a morphism for the operator (cf the example of `addz` in Section 1.2).

Then we want to show elementary properties on the lifting, and those can often be derived from the properties of the initial operation. Thanks to the compatibility lemma, it is easy to go back and forth between the operation and its lifting by making `pi` and the operation commute.

Example 5. We show that zero is a neutral element on the right for addition:

```
Definition zeroz := \pi_int (0, 0).
Lemma add0z x : addz zeroz x = x.
```

Then, by rewriting backwards with `reprK`, the statement of `add0z` is equivalent to:

```
addz (\pi_int (0, 0)) (\pi_int (repr x)) = (\pi_int (repr x))
```

which can be solved by rewriting backwards with `addz_compat`.

However, when faced with more complex expressions involving lifted operators, it becomes more complicated to control where rewriting must happen. In order to save the user from the need to use a chain of rewriting rules of the form `[pattern]op_compat`, we introduce an automated mechanism to globally turn an expression on the quotient into an expression on the base type. For this, an operation `Op` (respectively a function `F`) which is a lifting has to be recognized automatically. We must register in some way that the value associated with `(Op (\pi_Q x)(\pi_Q y))` is `(\pi_Q (op x y))` (respectively, that the value associated with `(F (\pi_Q x)(\pi_Q y))` is `(f x y)`). For this purpose, we define a structure `equal_to u`, the type of all elements of `Q` that are equal to `(u : Q)`:

```
Record equal_to Q u := EqualTo {equal_val : Q; _ : u = equal_val}.
Notation "{pi a}" := (equal_to (\pi a)) : quotient_scope.
Lemma piE (Q : Type) (u : Q) (m : equal_to u) : equal_val m = u.
```

The type parameter `u` of the structure is the translation to infer, while the content of the field `equal_val` is the information present in the goal. We also introduce a notation `{pi a}` for the type of all elements of `Q` which are equal to `(\pi a)`. Rewriting with lemma `piE`, will trigger an instantiation of the left hand side of the equality, which will infer a value for `m`, and thus for its type, which contains `u`.

To declare an instance for an operator `op`, we must provide a lifted operator `Op` and a proof that given two elements that are the canonical surjections of `x` and `y`, it returns a value which is the canonical surjection of `(op x y)`.

```
Canonical Structure op_equal_to (x y : T) (qT : quotType T)
   (X : {pi x}) (Y : {pi y}) : {pi (op x y)} :=
  EqualTo (Op X Y) (_ : pi qT (op x y) = Op X Y).
```

We declare that any term of the form `(\pi x)` has a trivial `{pi x}` structure (where both `u` and `equal_val` are `(\pi x)`):

```
Canonical Structure equal_to_pi T (qT : quotType T)
  (x : T) : {pi x} := EqualTo (\pi x) (erefl _).
```

Example 6. For the addition in `int`, we can write:

```
Lemma addz_pi (x y : nat * nat) (X : {pi x}) (Y : {pi y}) :
  \pi_int (add x y) = addz (equal_val X) (equal_val Y).

Canonical Structure addz_equal_to (x y : nat * nat)
   (X : {pi x}) (Y : {pi y}) : {pi (add x y)} :=
  EqualTo (addz (equal_val X) (equal_val Y)) (addz_pi X Y).
```

By declaring `addz_equal_to` as canonical, we can now use `piE` to rewrite an expression of the form `(addz` $\tilde{x}$ $\tilde{y}$`))` into `(\pi_int (add x y))`, where $\tilde{x}$ and $\tilde{y}$ are arbitrarily complicated expressions that can be canonically recognized as elements of respectively `{pi x}` and `{pi y}`, i.e. as being canonically equal to respectively `(\pi_int x)` and `(\pi_int y)`.

In the examples above, we defined by hand the quotient `int` of pairs of natural numbers `nat * nat`. Moreover, the equivalence was left implicit in the code. We could expect a generic construction of this quotient by the equivalence relation mentioned in the first example. We now deal with this deficiency.

1.5 Recovering Structure

This interface for quotient does not require the base type or the quotient type to be discrete (i.e. to have decidable equality), a choice type (see Section 2.1) or an algebraic structure. However, in the special case where the base type is discrete (respectively a choice type), we provide a mechanism to get the decidable equality structure (respectively the choice structure) on the quotient type.

Preserving structure through quotienting is more difficult to do for algebraic structures, but we provide such a support for quotients by an ideal, which we do not detail in this paper for the sake of space. Given a ring `R` with an ideal `I`, one can form the quotient `{ideal_quot I}` which has again a ring structure. Moreover, in that case, the canonical surjection `pi` is a ring morphism.

2 Quotient by an Equivalence Relation

Until now we have shown a quotient interface with no equivalence in its signature, and a notion of equivalence which is defined from the quotient. Now, we

explain how to get the quotient structure of a type by an equivalence relation. Given a type `T` and an equivalence relation `equiv`, we have to find a data type representing the quotient *i.e.* such that each element is an equivalence class.

A natural candidate to represent equivalence classes is the Σ-type of predicates that characterize a class. The elements of a given equivalence class are characterized by a predicate `P` that satisfies the following `is_class` property:

```
Definition is_class (P : T -> Prop) : Prop :=
  exists x, (forall y, P y <-> equiv x y).
```

Thus, we could define the quotient as follows.

```
Definition quotient := {P : T -> Prop | is_class P}.
```

However, because Coq equality is intensional, two predicates which are extensionally equal (*i.e.* equal on every input) may not be intensionally equal, and also, the proof that a given predicate is a class is not unique either. Hence, in order for `quotient` to have only one element by equivalence class, it would suffice to have both propositional extensionality and proof irrelevance for the sort `Prop`.

However, this is not enough to make `quotient` a quotient type. We have enough information to build the canonical surjection `pi`, but we do not know how to select a representative element for each class.

2.1 Quotient of a Choice Structure

If a type `T` has a choice structure [6], there exists an operator

```
xchoose : forall P : T -> bool, (exists y : T, P y) -> T.
```

which given a proof of `exists y, P y` returns an element `z`, such that `z` is the same if `xchoose` is given a proof of `exists y, Q y` when `P` and `Q` are logically equivalent.

Given a base type `T` equipped with a choice structure and a decidable equivalence relation `(equiv : T -> T -> bool)`, it becomes possible to build a quotient type. The construction is slightly more complicated than above.

For each class we can choose an element `x` in a canonical fashion, using the following `canon` function:

```
Lemma equiv_exists (x : T) : exists y, (equiv x) y.
Proof. by exists x; apply: equiv_refl. Qed.

Definition canon (x : T) := xchoose (equiv_exists x).
```

We recall that `xchoose` takes a proof of existence of an element satisfying a predicate (here the predicate `(equiv x)`) and returns a witness which is unique, in the sense that two extensionally equal predicates lead to the same witness. This happens for example with the two predicates `(equiv x)` and `(equiv y)` when `x` and `y` are equivalent: the choice function will return the same element `z`

which will be equivalent both to `x` and `y`. Such a canonical element is a unique representative for its class.

Hence, the type formed with canonical elements can represent the quotient.

```
Record equiv_quot := EquivQuot {
  erepr : T;
  erepr_canon : canon erepr == erepr
}.
```

Indeed, thanks to Boolean proof irrelevance [9], the second field (`erepr_canon`) of such a structure is unique up to equality, which makes this Σ-type a sub-type.

The representative function is trivial as it is exactly the projection `erepr` on the first field of the Σ-type `equiv_quot`. However, more work is needed to build the canonical surjection. Indeed we first need to prove that `canon` is idempotent.

```
Lemma canon_id (x : T) : canon (canon x) == canon x.
Definition epi (x : T) := EquivQuot (canon x) (canon_id x).
```

Finally, we need to prove that the canonical surjection `epi` cancels the representative `erepr`:

```
Lemma ereprK (u : equiv_quot T) : epi (erepr u) = u.
```

The proof of `ereprK` relies on the proof irrelevance of Boolean predicates.

Proof. Two elements of `equiv_quot` are equal if and only if their first projection `erepr` are equal, because the second field `erepr_canon` of `equiv_quot` is a Boolean equality, and has only one proof. Thanks to this `(epi (erepr u))` and `u` are equal if and only if `(erepr (epi (erepr u))` is equal to `(erepr u)`. But by definition of `epi`, `(erepr (epi (erepr u))` is equal to `(canon (erepr u))`, and thanks to the property `(erepr_canon u)`, we get that `(canon (erepr u))` is equal to `(erepr u)`, which concludes the proof.

We then package everything into a quotient structure:

```
Definition equiv_quotClass := QuotClass ereprK
Canonical Structure equiv_quotType := QuotType equiv_quot
    equiv_quotClass.
```

We declare this structure as canonical, so that any quotient by an equivalence relation can be recognized as a canonical construction of quotient type.

However, we omitted to mention that the proof of `canon_id` and hence the code of `epi` requires a proof that `equiv` is indeed an equivalence relation. In order to avoid to add unbundled side conditions ensuring `equiv` is an equivalence relation, we define an interface for equivalence relations which coerces to binary relations:

```
Structure equiv_rel := EquivRelPack {
  equiv_fun :> rel T;
        _ : reflexive equiv
        _ : symmetric equiv
        _ : transitive equiv
}.
```

The whole development about `equiv_quot` takes `(equiv : equiv_rel T)` as a parameter.

Remark 3. Given an equivalence relation `equiv` on a choice type `T`, we introduced the notation `{eq_quot equiv}` to create a quotient by inferring both the equivalence structure of `equiv` and the choice structure of `T` and applying the latter construction.

We recall that we defined the equivalence induced by the quotient by saying `x` and `y` are equivalent if `(\pi_Q x = \pi_Q y)`, where `Q` is the quotient type. We refine the former notation for equivalence modulo `Q` to specialize it to quotients by equivalence, as follows.

```
Notation "x = y %[mod_eq equiv]" := (x = y %[mod {eq_quot equiv}]).
```

In the present situation, it seems natural that this induced equivalence coincides with the equivalence by which we quotiented.

```
Lemma eqmodP x y : reflect (x = y %[mod_eq equiv]) (equiv x y).
```

Example 7. Let us redefine once again `int` as the quotient of $\mathbb{N} \times \mathbb{N}$ by the equivalence relation $((n_1, n_2) \equiv (m_1, m_2))$ defined by $(n_1 + m_2 = m_1 + n_2)$.

In this second version, we directly perform the quotient by the relation, so we first define the equivalence relation.

```
Definition equivnn (x y : nat * nat) := x.1 + y.2 == y.1 + x.2.

Lemma equivnn_refl : reflexive equivnn.
Lemma equivnn_sym : symmetric equivnn.
Lemma equivnn_trans : transitive equivnn.

Canonical Structure equivnn_equiv : equiv_rel (nat * nat) :=
  EquivRel equivnn equivnn_refl equivnn_sym equivnn_trans.
```

Then `int` is just the quotient by this equivalence relation.

```
Definition int := {eq_quot equivnn}.
```

This type can be equipped with a quotient structure by repackaging the quotient class of `equiv_quotType equivnn_equiv` together with `int`.

2.2 Quotient of Type with an Explicit Encoding to a Choice Type

In Section 3.4, we quotient by a decidable equivalence a type which is not a choice type, but has an explicit encoding to a choice type. Let us first define what we mean by explicit encoding, and then show how to adapt the construction of the quotient.

Definition 3 (Explicit encoding). *We say a type `T` with a equivalence relation `equivT` is explicitly encodable to a type `C` if there exists two functions `(T2C : T -> C)` and `(C2T : C -> T)` such that the following* coding property *holds:*

```
forall x : T, equivT (C2T (T2C x)) x.
```

The function `T2C` *is called the* encoding *function because it codes an element of* `T` *into* `C`*. Conversely, the function* `C2T` *is called the* decoding *function.*

Remark 4. Here `T` can be seen as a setoid, and the coding property can be interpreted as: `C2T` is a left inverse of `T2C` in the setoid `T`. It expresses that encoded elements can be decoded properly.

There is no notion of equivalence on the coding type `C` yet, but we can provide one using the equivalence induced by `T`. Thus, we define `equivC` by composing `equivT` with `C2T`.

```
Definition equivC x y := equivT (C2T x) (C2T y).
```

When `equivT` is a Boolean relation, so is `equivC`. When `C` is a choice type and `equivC` is a decidable equivalence on this choice type, we can reproduce the exact same construction of quotient as in Section 2.1, so that we get `ereprC : equiv_quot -> C`, `epiC: C -> equiv_quot` and a proof of cancellation `ereprCK`. Now we can compose these operators with `T2C` and `C2T`.

```
Definition ereprT (x : equiv_quotient) : T := C2T (ereprC x).
Definition epiT (x : T) : equiv_quotient := epiC (T2C x).
```

And we can prove the cancellation lemma `ereprTK` using `ereprCK` and the coding property.

```
Lemma ereprTK (x : equiv_quotient) : epiT (ereprT x) = x.
```

Finally, we have everything we need to create a quotient type, like in Section 2.1.

3 Applications

3.1 Rational Fractions

Given an integral domain D, i.e. a ring where $ab = 0 \Leftrightarrow a = 0 \vee b = 0$. One can build a field of rational fractions of D by quotienting $\{(x, y) \in D \times D \mid y \neq 0\}$ by the equivalence relation defined by:

$$(x, y) \equiv (x', y') := xy' = yx'$$

For example, rational numbers $\mathbb{Q}$ and the polynomial fractions $\mathbb{Q}(X)$ can be obtained through this construction.

In Coq, we first formalize the type `ratio` of pairs in a discrete ring, where the second element is non zero, and the above relation. Then, we prove it is an equivalence and quotient by it:

```
Inductive ratio := mkRatio { frac :> R * R; _ : frac.2 != 0 }.
Definition equivf x y := equivf_def (x.1 * y.2 == x.2 * y.1).
Canonical Structure equivf_equiv : equiv_rel.
Definition type := {eq_quot equivf}.
```

Remark 5. First restricting the domain to "ratios" and then quotienting is our way to deal with the partiality of the equivalence relation between pairs.

We then recover the decidability of the equality and the choice structure for free, through the quotient.[2] However, the ring and field structures cannot be derived automatically from the ring structure of $R \times R$ and have to be proven separately.

This development on rational fractions has been used in a construction of elliptic curves by Bartzia and Strub [10].

3.2 Multivariate Polynomials

The goal is to have a type to represent polynomials with an arbitrary number of indeterminates. From this we could start formalizing the theory of symmetric polynomials. Possible applications are another Coq proof of the Fundamental Theorem of Algebra [11] or the study of generic polynomials in Galois Theory.

We provide this construction for a discrete ring of coefficients R and a countable type of indeterminates. We form the quotient of the free algebra generated by constants in R, addition, multiplication and indeterminates (X_n), which we quotient by the relation defined by $t_1 \equiv t_2$ if t_1 and t_2 represent the same univariate polynomial in $(\dots(R[X_1])[X_2]\dots)[X_n]$ where n is big enough [12].

In order to know if two terms represent the same univariate polynomial in $(\dots(R[X_1])[X_2]\dots)[X_n]$ with n fixed, we iterate the univariate polynomial construction `{poly R}` from the SSReflect library.

3.3 Field Extensions

The proof of the Feit-Thompson Theorem relies on Galois theory and on a construction of the algebraic closure for countable fields (including finite fields). Both these constructions involve building a field extension through quotienting, but in two different ways. Although we did not develop the formal proof of these constructions, we briefly mention how quotients were used.

Galois theory studies the intermediate fields between a base field F and a given ambient field extension L. In this context, we build the field extension of F generated by the root $z \in L$ of a polynomials $P \in F[X]$ of degree n. It amounts in fact to building a subfield $F[z]$ of dimension n between F and L. This is done by quotienting the space $K_n[X]$ of polynomials with coefficients in K of degree at most n by the equivalence relation defined by $P \equiv Q := (P(z) = Q(z))$.

To build the algebraic closure of a *countable* field F, one can iterate the construction of a field extension by all irreducible polynomials of $F[X]$. Each iteration consists in building a maximal ideal and quotienting by it, which happens to be constructive because we have countably many polynomials.

[2] As there are two possible equality operators available (one obtained from the equality of the base type, the other from the equivalence relation), we must be very careful to let the user choose the one he wants, once and for all.

3.4 Real Algebraic Numbers

Finally, we used a quotient with an explicit encoding to a choice type (see Section 2.2) to build the real closed field of real algebraic numbers [13,7].

4 Related Work on Quotient Types

Given a base type (`T : Type`) and an equivalence (`equiv : T -> T -> Prop`), the Coq interface below is due to Laurent Chicli, Loïc Pottier and Carlos Simpson [3], following studies from Martin Hofmann [14]. It sums up the desired properties of a the quotient type: its existence, a surjection from the base type `T` to it, and a way to lift to the quotient functions that are compatible with the equivalence `equiv`.

```
Record type_quotient (T : Type) (equiv : T -> T -> Prop)
 (Hequiv : equivalence equiv) := {
 quo :> Type;
 class :> T -> quo;
 quo_comp : forall (x y : T), equiv x y -> class x = class y;
 quo_comp_rev : forall (x y : T), class x = class y -> equiv x y;
 quo_lift : forall (R : Type) (f : T -> R),
    compatible equiv f -> quo -> R;
 quo_lift_prop :
    forall (R : Type) (f : T -> R) (Hf : compatible equiv f),
    forall (x : T), (quot_lift Hf \o class) x = f x;
 quo_surj : forall (c : quo), exists x : T, c = class x
}.
```

where `\o` is the infix notation for functional composition and where `equivalence` and `compatible` are predicates meaning respectively that a relation is an equivalence (reflexive, symmetric and transitive) and that a function is constant on each equivalence class. We believe[3] they are defined as below:

```
Definition equivalence (T : Type) (equiv : T -> T -> Prop) :=
  reflexive equiv /\ symmetric equiv /\ transitive equiv.
Definition compatible (T R : Type) (equiv : T -> T -> Prop)
   (f : T -> R) := forall x y : T, equiv x y -> f x = f y.
```

Once this `type_quotient` defined, they [3] add the existence of the quotient as an axiom.

```
Axiom quotient : forall (T : Type) (equiv : T -> T -> Prop)
   (p:equivalence R), (type_quotient p).
```

Although this axiom is not provable in the type theory of Coq, its consistency with the Calculus of Constructions has been proved in [14]. The construction of this interface was made in order to study the type theory of Coq augmented

[3] No definitions for `equivalence` or `compatible` are explicitly given in [3].

with quotient types. This is not our objective at all. First, we want to keep the theory of Coq without modification, so quotient types do not exist in general. Second, we create an interface to provide practical tools to handle quotients types that do exist.

The reader may notice that here the field `quo` plays the role of our `quot_sort` and `class` the role of `pi` of our interface. The combination of `repr` and `reprK` is a skolemized version of `quo_surj`.

Remark 6. This is not exactly the case, because `quot_surj` is a `Prop` existential, which unlike existentials in `Type` cannot be extracted to a function `repr` which has the property `reprK`. This was already observed [3] in the study of the consistency of Coq with variants of `type_quotient`.

However, the parameters about `equiv` and properties about the lifting of morphism disappear completely in our interface, because they can all be encoded as explained in Section 1.2.

Example 8. For example, `quo_lift` can be encoded like this:

```
Definition new_quo_lift (T R : Type) (qT : quotType T)
  (f : T -> R) (x : Q) := f (repr x)
```

Note that the precondition `(compatible equiv f)` was not needed to define the lifting `new_quo_lift`. Only the property `quo_lift_prop` still needs the precondition.

Our approach can also be compared to Normalized Types [15]. The function `pi` can be seen as a user defined normalization function inside Coq.

5 Conclusion

This framework for quotient types is not a substitute for setoids and especially not for setoid rewriting. Indeed, it is not designed to make it easy to rewrite modulo equivalence, but to rewrite directly using equality inside the quotient type. Quotient types can mimic to some extend the behaviour of quotients in set-based mathematics.

This framework has already been useful in various non trivial examples. It handles quotients by ideals in the sense of the SSReflect library. Also, a natural continuation would be to study quotients of vector spaces, algebras and modules.

Also, in the SSReflect library, quotients of finite groups [16] are handled separately, taking advantage of the support for finite types. Maybe they could benefit from a connection with this more generic form of quotient.

Finally, it seems that this framework could work to quotient lambda-terms modulo alpha-equivalence. I did not attempt to do this construction, but it seems worth a try.

Acknowledgements. I would like to thank early experimenters of my framework, namely Pierre-Yves Strub and Russell O'Connor, who dared using it in real life formalizations and reported the problems they encountered. I also thank Georges Gonthier, for even earlier discussions on quotient types, thank to which I initially learned my way through the use of Canonical Structures in the SSReflect library. I also wish to thank the anonymous reviewers for their constructive feedback.

References

1. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming 13(2), 261–293 (2003); Special Issue on Logical Frameworks and Metalanguages
2. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning 2(1), 41–62 (2009)
3. Chicli, L., Pottier, L., Simpson, D.: Mathematical quotients and quotient types in Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 95–107. Springer, Heidelberg (2003)
4. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. INRIA Technical report, `http://hal.inria.fr/inria-00258384`
5. The Mathematical Components Project: SSReflect extension and libraries, `http://www.msr-inria.inria.fr/Projects/math-components/index_html`
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. Cohen, C.: Formalized algebraic numbers: construction and first order theory. PhD thesis, École polytechnique (2012)
8. Saibi, A.: Typing algorithm in type theory with inheritance. In: Proceedings of Principles of Programming Languages, POPL 1997, pp. 292–301 (1997)
9. Hedberg, M.: A coherence theorem for Martin-Löf's type theory. Journal of Functional Programming, 4–8 (1998)
10. Bartzia, I., Strub, P.Y.: A formalization of elliptic curves (2011), `http://pierre-yves.strub.nu/research/ec/`
11. Cohen, C., Coquand, T.: A constructive version of laplace's proof on the existence of complex roots. Journal of Algebra 381, 110–115 (2013)
12. Cohen, C.: Reasoning about big enough numbers in Coq (2012), `http://perso.crans.org/cohen/work/bigenough/`
13. Cohen, C.: Construction of real algebraic numbers in Coq. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 67–82. Springer, Heidelberg (2012)
14. Hofmann, M.: Extensional concepts in intensional type theory. Phd thesis, University of Edinburgh (1995)
15. Courtieu, P.: Normalized types. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 554–569. Springer, Heidelberg (2001)
16. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA (2007)

Mechanical Verification of SAT Refutations with Extended Resolution

Nathan Wetzler, Marijn J.H. Heule, and Warren A. Hunt Jr.*

The University of Texas at Austin

Abstract. We present a mechanically-verified proof checker developed with the ACL2 theorem-proving system that is general enough to support the growing variety of increasingly complex satisfiability (SAT) solver techniques, including those based on extended resolution. A common approach to assure the correctness of SAT solvers is to emit a proof of unsatisfiability when no solution is reported to exist. Contemporary proof checkers only check logical equivalence using resolution-style inference. However, some state-of-the-art, conflict-driven clause-learning SAT solvers use preprocessing, inprocessing, and learning techniques, that cannot be checked solely by resolution-style inference. We have developed a mechanically-verified proof checker that assures refutation clauses preserve satisfiability. We believe our approach is sufficiently expressive to validate all known SAT-solver techniques.

1 Introduction

Satisfiability (SAT) solvers are becoming commonplace for a variety of applications, including model checking [1], equivalence checking, hardware verification, software verification, and debugging. These tools are often used not only to find a solution for a Boolean formula, but also to make the claim that no solution exists. If a solution is reported for a given formula, one can check the solution linearly in the size of the formula. But when no solution is reported to exist, we want to be confident that a SAT solver has fully exhausted the search space. This is complicated by the fact that state-of-the-art solvers employ a large array of complex techniques to maximize efficiency. Errors may be introduced at a conceptual level as well as a implementation level. Formal verification, then, is a reasonable approach to detect errors or to assure that results produced by SAT solvers are correct.

One method of assurance is to apply formal verification to the SAT solver itself. This involves modeling a SAT solver, specifying the desired behavior, and using a tool—such as a theorem prover—to show that the model meets its specification. The benefit of such a direct approach is that the solver would only need to run once for a given input. There are many problems with this approach, however. SAT solvers are constantly evolving, and each new implementation would require a new proof. Furthermore, it is hard to balance verification requirements with efficiency. Lescuyer and Conchon [2] formalized and verified the basic Davis-Putnam-Logemann-Loveland (DPLL) [3,4] algorithm

* The authors are supported by the National Science Foundation under grant CNS-0910913 and DARPA contract number N66001-10-2-4087.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 229–244, 2013.

using Coq [5]. Shankar and Vaucher [6] verified a DPLL solver using PVS. Marić [7,8] verified pseudocode fragments of a conflict-driven clause-learning (CDCL) [9] solver in 2009 and verified a CDCL-style solver using Isabelle/HOL [10] in 2010. Oe et al. [11] provided an verified CDCL-style solver in Guru. Several key techniques, such as clause minimization during conflict analysis, have yet to be mechanically verified.

Another approach is to validate the output of a SAT solver. A proof trace is a sequence of clauses that are claimed to be redundant with respect to a given formula. If a SAT solver reports that a given formula is unsatisfiable, it can provide a proof trace that can be checked by a smaller, trusted program called a proof checker. A refutation is a verified proof trace containing the empty clause. Ideally, a proof trace should be compact, easy to obtain, efficient to verify, and should facilitate a simple checker implementation. Moreover, we "only" need to formally verify the proof checker. By focusing verification efforts on a proof checker, we gain assurance while avoiding the need to verify a variety of solvers with differing implementations.

Proof traces have traditionally established redundancy in the form of resolution chains [12,13,14]. In resolution-style proofs, clauses are iteratively added to a formula provided that they can be derived from a sequence of applications of the resolution rule. Resolution proof traces are simple to express and can be efficiently validated, but they tend to be enormous and difficult to obtain from a solver. Weber [15,16] demonstrated the first mechanically-verified resolution-based proof checker using Isabelle/HOL. Darbari et al. [17] verified a resolution-based proof checker in Coq which is able to execute outside of the theorem-prover environment. Armand et al. [18] extended a SAT resolution-based proof checker to include SMT proofs using Coq.

Alternatively, one can use unit propagation, one of the basic SAT simplification techniques, to check proof traces. Each proof clause is shown to be redundant by adding the complement of the clause as unit clauses, performing unit propagation with respect to the conjunction of the original formula and all verified proof clauses so far, and then checking for a conflict. This process is known as reverse unit propagation (RUP) [13,19]. RUP proofs are compact and easy to obtain; however, RUP checkers are somewhat inefficient and more complicated than their resolution-based counterparts. Oe and Stump [20] implemented a non-verified RUP proof checker in C++ and proposed a verified RUP proof checker in Guru.

However, both resolution and RUP proof formats lack the expressivity to capture a growing number of techniques used in state-of-the-art SAT solvers. SAT solvers often use preprocessing and inprocessing in addition to (CDCL-style) learning, and some of these techniques cannot be expressed by resolution-style inference such as bounded-variable addition [21], blocked-clause addition [22], and extended learning [23]. These techniques can be expressed, however, by extended resolution (ER) [24] or a generalization of ER [22]. Järvisalo et al. [25] demonstrated a hierarchy of redundancy properties, the most expressive of which is Resolution Asymmetric Tautology (RAT), which is a generalization of RUP. All preprocessing, inprocessing, and learning techniques used in contemporary solvers can be expressed by the addition and removal of RAT clauses. One key difference, however, between RAT and RUP (or resolution) is that RAT checks satisfiability equivalence instead of logical equivalence.

In [26], we proposed a new proof format based on the RAT redundancy property and described a fast implementation of a proof checker for this format written in C. In this paper, we present a mechanically-verified SAT proof checker using ACL2 [27] based on the RAT redundancy property. This includes a mechanical proof of the redundancy (via satisfiability equivalence) of RAT clauses. Our implemented proof checker is the most expressive proof checker to date and it is mechanically verified.

In Section 2 we will introduce ACL2 and formalize key SAT concepts including unit propagation and resolution. We will also discuss a redundancy hierarchy and provide our implementation of RAT and our RAT proof checker. We will give a specification for our RAT proof checker in Section 3, and present our mechanical proof of correctness in Section 4. Finally, we conclude in Section 5.

2 Formalization

2.1 ACL2

We used the ACL2 system [27] to develop our formalization, specification, and proof. ACL2 is a freely-available system that provides a theorem prover and a programming language, both of which are based on a first-order logic of recursive functions. The logic is compatible with Common Lisp—indeed, "ACL2" is an acronym that might be written as "ACL^2" and stands for "A Computational Logic for Applicative Common Lisp"—and an executable image can be built on a number of Common Lisp implementations. ACL2 provides efficient execution by way of Common Lisp compilers.

The initial theory for ACL2 contains axioms for primitive functions such as `cons` (the constructor for an ordered pair), `car` (the head of a list or first component of a pair), and `cdr` (the tail of a list or second component of a pair). It also contains axioms for Common Lisp functions, such as `member`, and it introduces axioms for user-supplied definitions. ACL2 provides a top-level read-eval-print loop. Arbitrary ACL2 expressions may be submitted for evaluation. Of special interest are events, including definitions and theorems; these modify the the theorem prover's logical database for subsequent proof and evaluation. Function definitions are typically expressed using the `defun` event and theorems using the `defthm` event. We call an ACL2 function a *predicate* if it returns a Boolean value.

As is the case for Lisp, the syntax of ACL2 is generally case-insensitive and is based on prefix notation: `(function argument1 ... argumentk)`. For example, the term denoting the sum of `x` and `y` is `(+ x y)`. A semicolon "`;`" starts a comment. ACL2 also supports the functions `let` and and `let*` for parallel and sequential bindings, respectively. Functions may return mulitple values using the constructor `mv` which stands for "multiple value". Elements of `mv` may be accessed with the function `mv-nth` which accesses the nth value of an `mv`. The function `mv-let` takes three arguments: a list of bindings, a function that returns an `mv` (with the same number of values as the bindings), and a body. For example, suppose that a function `f` returns two values and we wish to compute their sum. We can compute this with the following term:

```
(mv-let (x y)
        (f ...)
        (+ x y))
```

ACL2 does not have native support for quantification in the logic; however, ACL2 allows a user to define *Skolemized functions* using the `defun-sk` event. This event introduces a witness function that will return a witness object if such an object exists. For example, if we wanted to express the mathematical statement, "there exists an x such that $x < y$", we could do so with the event:

```
(defun-sk exists-x-<-y (y) (exists x (< x y)))
```

This event defines a non-executable, one-argument function `exists-x-<-y-witness` that will return an `x` if one exists. The non-executable function `exists-x-<-y` returns true if `exists-x-<-y-witness` finds such an object.

Links to papers that apply ACL2, as well as detailed hypertext documentation and installation instructions, may be found on the ACL2 home page.[1]

2.2 Satisfiability Basics

We will now begin introducing some SAT concepts. We will forego providing the traditional SAT notation and will instead use ACL2 notation. In this way, we can define key SAT terminology while describing our formalization.

We model Boolean variables with positive integers. For a Boolean variable `v`, there are two *literals*, the positive literal `l` and the negative literal computed by `(negate l)`. We represent positive and negative literals as positive and negative integers, and we recognize them with the predicate `literalp`. A *clause* is a finite disjunction of literals, and a clause is a *tautology* if it contains the conflicting literals `l` and `(negate l)` for some `l`. We introduce the predicate `no-conflicting-literalsp` to recognize non-tautological lists, and we define the predicate `clausep` to recognize non-tautological ACL2 lists of unique literals.[2] A *conjunctive normal form* (CNF) *formula*, recognized by predicate `formulap`, is a finite conjunction of clauses which we represent as an ACL2 list of non-tautological clauses. We do not require clauses to be unique within a formula. In the rest of this paper, we will assume all formulas to be in CNF. The set of literals occurring in a formula `f` is computed by the function `all-literals`.

A truth *assignment* for a formula `f` is a partial function that maps literals `l` in `(all-literals f)` to Boolean values. We model an assignment, with predicate `assignmentp`, as a ACL2 list of unique non-conflicting literals (note that the implementations of `assignmentp` and a `clausep` are the same). In our ACL2 representation, we define special values `true`, `false`, and `undef` with corresponding predicates `truep`, `falsep`, `undefp`. The evaluation of literal `l` with respect to assignment `ta` is computed by the function `evaluate-literal` which returns `true` if `(member l ta)`, `false` if `(member (negate l) ta)`, and `undef` otherwise. The evaluation of a clause `c` with respect to assignment `ta` is computed by the function `evaluate-clause` which returns `true` if `(evaluate-literal l ta)` is `true` for some literal `l` in `c`, `false` if `(evaluate-literal l ta)` is `false` for all literals `l` in `c`, and `undef` otherwise. The evaluation of a formula `f` with respect to assignment `ta` is computed by the function `evaluate-formula` which returns `true` if `(evaluate-clause c ta)` is `true` for all clauses `c` in `f`, `false` if `(evaluate-clause c ta)` is `false` for some clause `c` in `f`, and `undef` otherwise.

[1] `www.cs.utexas.edu/users/moore/acl2/`

[2] This is the same representation as the SAT competition DIMACS format.

We say a clause `c`, or a formula `f`, is *satisfied* by an assignment `ta` if evaluation of `c`, or `f`, with respect to `ta` is `true`. An assignment that satisfies a formula is called a *solution*. We say a clause `c`, or a formula `f`, is *falsified* by an assignment `ta` if evaluation of `c`, or `f`, with respect to `ta` is `false`. A formula `f` is *satisfiable* if there exists a solution for `f` and *unsatisfiable* if there does not exist a solution for `f`. Two formulas are *logically equivalent* if and only if they have the same set of satisfying assignments. Two formulas are *satisfiability equivalent* if and only if they are both satisfiable or both unsatisfiable.

The negation of a clause is an assignment computed by `negate-clause`. For example, `(negate-clause '(1 -2 3))` returns the assignment `(-1 2 -3)`. The negation of an assignment is a clause computed by `negate-assignment`. Both functions share the same implementation and are complements of each other.

2.3 Proof Traces

Conflict-driven clause learning (CDCL) [9] is the leading paradigm of modern SAT solvers. A core aspect of CDCL solvers is the addition and removal of clauses. The main form of CDCL reasoning is known as conflict analysis, which adds *conflict clauses* encountered during search. Additionally, state-of-the-art CDCL solvers use preprocessing and inprocessing techniques that both add and remove clauses.

A clause `c` is *redundant* with respect to a formula `f` if `(cons c f)` is satisfiability equivalent to `f`. A formula is a set of clauses, and we write `(cons c f)` to extend a formula. We say the addition of a redundant clause `c` to `f` *preserves satisfiability*.

A *proof trace* is a sequence of clauses that are redundant with respect to an input formula. Note that a proof trace is a sequence because the order of the clauses in a proof trace is essential. As an example, two clauses `c1` and `c2` may both be redundant with respect to a formula `f`, but `c2` may not be redundant with respect to the extended formula `(cons c1 f)`. A proof trace can be validated by a *proof checker* tool that iteratively (or recursively) removes the first clause `c` from a proof trace, checks the clause `c` for redundancy with respect to the current formula `f`, and then extends the set `f` with `c`. A validated proof trace that contains the empty clause, which cannot be satisfied, is called a *refutation*.

2.4 Resolution and Resolution Proofs

The early approaches to verify proof traces were based on resolution [12]. The *resolution rule* states that given a clause `c1` containing literal `l` and a clause `c2` containing literal `(negate l)` that the *resolvent* is the union of `c1` without `l` and `c2` without `(negate l)`. The resolvent is logically implied by `c1` and `c2`. We compute resolvents with the function `(resolution l c1 c2)`. Note that one can compute resolution without the use of a resolving literal, i.e. `l`. We chose this form in our model because it is more explicit and eases proof burdens.

Example 1. The clauses `(1 -2)` and `(2 -3)` contain a conflicting literal, i.e, one clause contains `l` and the other contains `(negate l)`. Therefore, we can apply resolution on them. `(resolution -2 '(1 -2) '(2 -3))` results in `(1 -3)`.

Resolution proof traces are a sequence of clauses whose redundancy can be established by a sequence of resolutions on clauses from an input formula. Clauses added by CDCL solvers can be simulated by a sequence of resolutions [28]. Because resolution is such an elementary operation, simple and fast checking algorithms exist [12] that assure that a trace of resolution applications is correct. However, resolution proofs tend to be very large, and it may be hard to modify a SAT solver to emit a resolution refutation; for instance, one must determine the clauses on which to apply resolution, and specifying the order of resolutions can be difficult.

2.5 Extended Resolution

For a given formula `f`, the *extension rule* [24] allows one to iteratively add clauses to `f` encoding the logical AND of two existing literals as a new Boolean variable. More specifically, given variables `1`, `2` that appear in `f` and a variable `3` which does not appear in `f`, the clauses `((3 -1 -2) (-3 1) (-3 2))` can be added to `f`. *Extended Resolution* (ER) [24] is a proof system in which the extension rule is repeatedly applied to a formula `f`, followed by applications of the resolution rule. This proof system surpasses what can be expressed using only resolution.

ER [24] is the basis for some techniques used during learning [23] and preprocessing [21] in state-of-the-art SAT solvers. Refutations using ER can be exponentially smaller than refutations based solely on resolution. Examples include the pigeon-hole problems where Haken [29] showed that all resolution proofs are exponential in size, while Cook [30] demonstrated that some ER proofs can be polynomial in size. Our proof checker supports techniques that are based on ER.

2.6 Unit Propagation and Clausal Proofs

Goldberg and Novikov [19] proposed an alternative to resolution-based proofs. They observed that each clause added by CDCL conflict analysis can be checked using *unit propagation*, also known as *Boolean constraint propagation*. This technique simplifies a formula `f` based on unit clauses. A clause of any length is *unit* if all literals in the clause `c` evaluate to `false` under an assignment `ta` except for one, which evaluates to `undef`; this literal is called a *unit literal* and is added to `ta`. Adding the unassigned literal to `ta` makes `c` evaluate to `true` under the extended assignment. This procedure continues until a unit clause cannot be found.

Unit propagation can be used to check if a clause `c` is logically implied by a formula `f`. Start with the assignment `(negate-clause c)`. Apply unit propagation until a *conflict* arises, i.e., some clause in `f` is falsified. If a conflict occurs, then adding `c` to `f` preserves logical equivalence [19]. Clauses that can be checked using this procedure are also known as *reverse unit propagation* (RUP) clauses [13].

Example 2. Consider the formula `f = ((-1 2) (-2 -3) (3 4))` and the assignment `ta = (1 -4)`. Formula `f` under `ta` contains two unit clauses: `(-1 2)` with unit literal `2` and `(3 4)` with unit literal `3`. Extending `ta` with the unit literals results in the extended assignment `(1 2 3 -4)`. The extended assignment falsifies clause `(-2 -3)`, so unit propagation results in a conflict. Unit propagation on `f` under `(1 -4)` results in a conflict, which shows that clause `(-1 4)` is redundant with respect to `f`.

Because unit propagation will play a key role in our proof checker, we formalize this technique below. The function `(is-unit-clause c ta)` returns the unit literal if one exists or `nil` if `c` is not unit. The function `(find-unit-clause f ta)` recursively checks if each clause in `f` is unit, returns the multiple-value pair containing the unit literal and unit clause if a unit clause exists, and returns the multiple-value pair `(mv nil nil)` otherwise. Unit propagation for a formula `f` with respect to assignment `ta` is defined as follows:

```
(defun unit-propagation (f ta)           ;; Formula f, assignment ta
  (declare (xargs :measure (num-undef f ta))) ;; Termination measure
  (mv-let (ul uc)                        ;; Unit literal, unit clause
          (find-unit-clause f ta)        ;; Found by find-unit-clause
          (declare (ignorable uc))       ;; Unit clause not needed
          (if (not ul)                   ;; No unit literal?
              ta                         ;; Then, return assignment
            (unit-propagation f (cons ul ta))))) ;; Recur with new ta
```

The `mv-let` (Section 2.1) calls `find-unit-clause`, binds the results to `ul` and `uc`, declares that `uc` will be ignored, and executes the body which tests for a unit literal and recurs with an extended assignment if a unit literal is found.

ACL2 proves the termination of each function admitted to the logic. In most instances, ACL2 will be able to prove the termination of a function without additional help, but sometimes one might need to explicitly provide a measure. A measure is a function which computes a value that must decrease (with respect to a well-defined relation) during every recursive call. We provide a measure for `unit-propagation` in the definition above called `num-undef` that computes the number of clauses in `f` that evaluate to `undef` under `ta`. While `ta` is recursively *extended* during `unit-propagation`, the measure will *decrease* because each unit literal added to `ta` will make one `undef` clause become `true`.

2.7 Redundancy Hierarchy

There are many properties that can establish the redundancy of a clause. Järvisalo et al. [25] offers a hierarchy of fifteen redundancy properties that can be computed in polynomial time with respect to a formula. If a clause has one of those fifteen properties with respect to a given formula, then adding the clause to the formula preserves satisfiability. A discussion of all fifteen properties goes beyond the scope of this paper, so we will focus on the four properties that are related to our proof checker. A reduced hierarchy is shown in Fig 1.

We have already presented two redundancy properties. A clause has T (*tautology*) if and only if it contains the literals `l` and `(negate l)` for some `l`. A clause has AT (*asymmetric tautology*) if and only if reverse unit propagation results in a conflict. Note that any clause with property T trivially has the property AT. Many techniques used in SAT solvers can be expressed as a trace of clauses with AT including CDCL learning [31], variable elimination (DP resolution) [3,32] and subsumption. Adding clauses with T or AT to a formula preserves logical equivalence.

The other two redundancy properties are related to resolution. For a given literal `l` and a formula `f`, let `f-neg-l` denote the subset of clauses in `f` that contain the literal `(negate l)`. First, a clause `c` has RT (*resolution tautology*) if and only if `c` has T or it contains a literal `l` such that all resolvents between `c` and a clause in `f-neg-l` have T. Second, a clause `c` has RAT (*resolution asymmetric tautology*) if and only if `c` has AT or it contains a literal `l` such that all resolvents between `c` and a clause in `f-neg-l` have AT. If a clause has any redundancy property in the hierarchy, then it also has RAT [25]. Techniques that can be expressed using RAT but not with AT include blocked clause addition [22], bound variable addition [21], extended resolution [24], and extended learning [23].

Example 3. Let formula `f` be `((1 2) (2 3) (-2 -3))`.

- The clause `(1 -1)` is a tautology (has T) because it contains `1` and `(negate 1)`.
- The clause `(1 -3)` does not have T. However, it has RT (and RAT) with respect to `f` and literal `1`, because `f` contains no clauses with literal `-1`. Furthermore, it also has AT because unit propagation with the assignment `(-1 3)` results in a conflict.
- The clause `(-1 3)` has RAT, but not T, AT, or RT. Unit propagation under the assignment `(1 -3)` does not result in a conflict, so `(-1 3)` does not have AT. Also, `(-1 3)` does not have RT, because there are non-tautological resolvents with `(1 2)` and `(-2 -3)`. Finally, the only resolvent on literal `-1` is computed by resolving `(-1 3)` with `(1 2)` to obtain `(2 3)`, which is already in `f`. So, unit propagation on the negation of the resolvent, `(-2 -3)`, results in a conflict. Hence, `(-1 3)` has RAT.

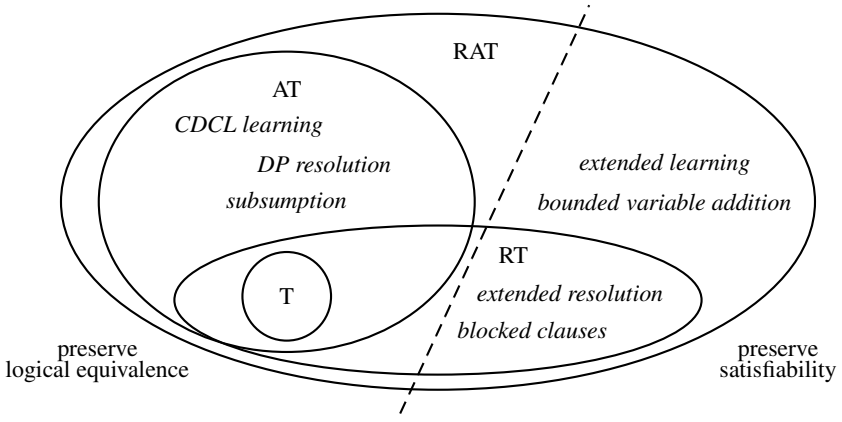


Fig. 1. Relationships between clause redundancy properties that can be computed in polynomial time with respect to the size of a formula. Techniques, shown in italics, are positioned based on the most efficient check to verify that technique. All techniques used in state-of-the-art SAT solvers can expressed as a sequence of RAT clauses [25]. The dashed line separates techniques that preserve logical equivalence and techniques that preserve satisfiability.

2.8 RAT

RAT is the strongest redundancy property in the hierarchy of [25] that preserves satisfiability. All preprocessing, inprocessing, and solving techniques in state-of-the-art SAT

solvers can be expressed in terms of addition and removal of RAT clauses [25]. Recall that a clause `c` has RAT if and only if `c` has AT or it contains a literal `l` such that all resolvents between `c` and a clause in `f-neg-l` have AT. A clause `c` has AT if unit propagation on the assignment `(negate c)` results in a conflict.

We define *resolution asymmetric tautology* (RAT) as follows:

```
(defun ATp (f c)  ;; Given formula f, clause c
  (falsep (evaluate-formula f
                            (unit-propagation f (negate-clause c)))))

;; Given clause list cl, formula f, clause c, and literal l
(defun RATp1 (cl f c l)
  (if (atom cl)                                ;; End of clause list?
      t                                        ;; Then, success
    (if (not (member (negate l) (car cl)))     ;; No resolution?
        (RATp1 (cdr cl) f c l)                 ;; Then, continue
      (let ((r (resolution l c (car cl))))     ;; Make resolvent
        (if (tautologyp r))                    ;; Resolvent has T?
            (RATp1 (cdr cl) f c l)             ;; Then, continue
          (and (ATp f r)                       ;; Resolvent has AT?
               (RATp1 (cdr cl) f c l)))))))    ;; Then recur, else fail

(defun RATp (f c l)  ;; Given formula f, clause c, and literal l
  (RATp1 f f c l))   ;; Copy f for recursion in helper function
```

The function `RATp` destructively recurs over a formula but needs a copy of the formula to compute asymmetric tautologies. Therefore, we begin by calling a helper function `RATp1` which has two copies of formula `f`. The first copy will be used for recursion and bound as `cl` while the second copy will remain untouched. If we have checked all clauses in `cl`, then we return `t` because `c` has RAT. If `(negate l)` is not a member of the current clause `(car cl)`, then we recur. We next perform the resolution of `c` and `(car cl)` on `l` and bind the result to `r`. Finally, we check if `r` is a tautology or if `r` has AT. If either of these is true, we recur, and we return `nil` otherwise.

2.9 Proof Checker

We now present the implemenation of our proof checker. Our checker works by ensuring that each clause `c` in a proof trace `pt` has `ATp` with respect to formula `f` or `RATp` with respect to formula `f` on the first literal of the clause. If `c` can be verified, then `c` is added to `f` before recurring.

```
(defun verify-clause (c f)        ;; Given clause c, formula f
  (or (ATp f c)                   ;; Verify by AT, OR
      (and (not (atom c))         ;; Check for non-empty clause, AND
           (RATp f c (car c)))))  ;; Verify by RAT w.r.t. 1st literal

(defun verify-proof (pt f)          ;; Proof trace pt, formula f
  (if (atom pt)                     ;; End of proof trace?
      t                             ;; Then, success
```

```
(if (verify-clause (car pt) f)  ;; First clause in pt verified?
    (verify-proof (cdr pt)            ;; Then, recur with
                  (cons (car pt) f))  ;; extended formula
  nil)))                          ;; Else, fail
```

To be clear, we assume that the first literal of the clause `(car c)` is the literal on which to perform resolution during the `RATp` check. This is a design choice and the efficiency of this method is described in [26]. Note that the empty clause `()` will fail the `(not (atom c))` case in `verify-clause`. We do this because `()` does not have an explicit resolution literal.

One can simply redefine `verify-clause` to only check clauses for `ATp`, removing the call to `RATp`. This creates a traditional RUP proof checker.

Example 4. Let formula `f = ((1 2 -3) (-1 -2 3) (2 3 -4) (-2 -3 4) (-1 -3 -4) (1 3 4) (-1 2 4) (1 -2 -4))`. A refutation for `f` is `((1) (2) ())`.

3 Specification

We will introduce a few new concepts and then state our main theorem. The function `clause-listp` recognizes lists of clauses, similar to `formulap`. We define a `proofp` object to be a clause list that has been checked by our proof checker. A proof is a `refutationp` object if it also contains the empty clause. The predicate `solutionp` recognizes assignments that satisfy a given formula.

```
(defun proofp (pt f)            ;; A proof is a clause sequence
  (and (clause-listp pt)        ;; that has been verified with
       (verify-proof pt f)))    ;; respect to a formula

(defconst *empty-clause* nil)   ;; The empty clause

(defun refutationp (p f)        ;; A refutation is a proof that
  (and (proofp p f)             ;; contains the empty clause
       (member *empty-clause* p)))

(defun solutionp (ta f)         ;; A solution is an assignment
  (and (assignmentp ta)         ;; that satisfies a formula
       (truep (evaluate-formula f ta))))
```

We use the `defun-sk` event (Section 2.1) to define the notion that there exists a solution for a formula.

```
(defun-sk exists-solution (f)
  (exists ta (solutionp ta f)))
```

With those definitions, we can state our main theorem.

```
(defthm main-theorem
  (implies (and (formulap f)            ;; Given a formula f
                (refutationp r f))      ;; And refutation r
           (not (exists-solution f))))  ;; Then f is unsatisfiable
```

This theorem reads that if given a refutation `r` and a formula `f`, then there does not exist a solution for `f`. In other words, a refutation that has been verified by our RAT checker implies that a formula is unsatisfiable. Note that this is only a specification for correctness of our proof checker as defined above.

4 Proof

We will now describe the mechanical proof of `main-theorem`. ACL2-style proofs are generally a sequence of `defthm` and `defun` events. While ACL2 processes events in a bottom-up fashion, we provide a top-down description of our ACL2 proof. We want to prove that a refutation `r` for a formula `f` implies that the formula is unsatisfiable. An outline of the proof is as follows:

1. We will prove the contrapositive—if there exists a solution `s` for `f`, then we should not be able to verify the refutation.
2. We prove that the empty clause cannot be redundant with respect to `f` provided `s` is a solution for `f`.
3. We show that every clause `c` in `r` is redundant. This contradicts (2) because the empty clause is a member of `r`. We use structural induction on `r` to prove this.
   (a) We show that clauses with `ATp` are redundant.
   (b) We show that clauses with `RATp` are redundant. We case split based on the result of `(evaluate-clause c s)`.
      i. If `(evaluate-clause c s)` is `true`, then `s` is a solution for `c`.
      ii. If `(evaluate-clause c s)` is `undef`, then we construct a new solution `s+` that consists of `s` with an `undef` literal in `c`.
      iii. If `(evaluate-clause c s)` is `false`, then we construct a new solution `s*` that is `s` with the exception that one literal in `s` has been negated.

In order to prove `main-theorem`, we first expand the definition of `refutationp` and contrapose the call of `verify-proof` with `exists-solution`. We will now use structural induction on the proof trace `pt`.

```
(defthm verify-proof-induction
  (implies (and (clause-listp pt)
                (formulap f)
                (exists-solution f)
                (member *empty-clause* pt))
           (not (verify-proof pt f))))
```

Recall that at each step `verify-proof` adds a clause from the proof trace `pt` to the formula. In our induction step, we will show that clauses in the proof with `ATp` or `RATp` are redundant. This allows us to derive a contradiction because `*empty-clause*` is a member of the refutation, does not have `ATp` or `RATp` with respect to a satisfiable formula, and is therefore not satisfiability equivalent.

```
(defthm *empty-clause*-lemma
  (implies (solutionp s f)
           (not (ATp f *empty-clause*))))
```

We prove this lemma with set reasoning. Specifically, if an assignment falsifies a formula, then a superset of that assignment will falsify a formula. Any solution must be a superset of the assignment constructed by performing unit propagation on the empty clause. Furthermore, the empty clause does not have `RATp` because there is no literal with which to perform resolution. We exclude this case by performing a `(not (atom c))` check in `verify-clause`.

We now return to the induction step of `verify-proof-induction`. This is a rather odd induction step because it needs to be expressed in terms of existentials. We will prove that if there exists a solution for the formula, then there exists a solution for the formula extended with a clause from the proof trace. In other words, we want to show that the extended formula is satisfiability equivalent to the original formula.

Here the proof diverges based on whether a proof clause has `ATp` or `RATp`. We consider the `ATp` case in Section 4.1 and the `RATp` case in Section 4.2.

4.1 ATp

If a clause `c` has `ATp` with respect to a formula `f`, then we will show that `(cons c f)` is logically equivalent to `f` (and therefore satisfiability equivalent to `f`).

```
(defthm ATp-lemma
  (implies (and (ATp f c)
                (exists-solution f)
                (formulap f)
                (clausep c))
           (exists-solution (cons c f))))
```

We expand `(exists-solution f)` to obtain a witness solution `s`. We will use `s` as a witness for the term `(exists-solution (cons c f))` in the conclusion. We know that `s` satisfies every clause in the original formula, so it is sufficient to show that `s` satisfies the `ATp` clause. Recall the definition of `ATp`. We replace the clause `c` with an abstraction `(negate-assignment ta)`. As previously stated, `negate-assignment` and `negate-clause` are complements of each other; `(negate-clause (negate-assignment ta))` simplifies to `ta`. We are then left with the following theorem.

```
(defthm ATp-lemma-induction
  (implies (and (falsep (evaluate-formula f (unit-propagation f ta)))
                (truep (evaluate-formula f s))
                (formulap f)
                (assignmentp ta)
                (assignmentp s))
           (truep (evaluate-clause (negate-assignment ta) s))))
```

We want to show that there is an `l` such that `l` is a member of `s` and `(negate l)` is a member of `ta`. Let assignment `up-ta` be the result of `(unit-propagation f ta)`. Because `up-ta` falsifies `f`, there must be a clause `c*` that is falsified by `up-ta`. Since `s` satisfies `f`, `s` also satisfies `c*`. Let `l*` be the literal that is a member of `c*` and `s`. Notice that `(negate l*)` is a member of `up-ta`.

We will induct on the extended assignment `up-ta`. In the base case, `up-ta` is equal to `ta`. Thus, `(negate l*)` is a member of `ta` and `l*` is a member of `s`. In the induction

step, up-ta is (cons ul ta) for some unit clause uc with unit literal ul. Again, there must be a clause c* that is falsified by up-ta but satisfied by s. Let l* be the literal that is a member of c* and s. Either (negate l*) is equal to ul or (negate l*) is in ta. If (negate l*) is in ta, then we are done. Otherwise, ul is equal to (negate l*), i.e. (negate ul) is in s. Consquently, uc was not satisfied by ul. All literals in uc not equal to ul are falsified by ta from the definition of unit clause. Let l** be the literal in s that satisfies uc. Since l** cannot be ul, (negate l**) is in ta and l** is in s.

Commentary. The induction for ATp-lemma-induction is the most difficult part of the proof of ATp-lemma. First, the induction is blocked by (negate-clause c). We tried several abstractions, all of which affected the goal (truep (evaluate-clause c s)), before settling on the use of negate-assignment. This abstraction lets us perform the correct induction without significantly changing the goal. Second, the induction itself is subtle because of the custom measure provided to unit-propagation. The assignment continues to grow during every recursive call of unit-propagation, but the number of undef clauses decreases.

4.2 RATp

We wish to show that if there exists a solution s for formula f and a clause c has RATp with respect to f and literal l in c, then there is a solution for the set (cons c f).

```
(defthm RATp-lemma
  (implies (and (formulap f)
                (clausep c)
                (member l c)
                (exists-solution f)
                (RATp f c l))
           (exists-solution (cons c f))))
```

Let assignment s satisfy f. Consider the cases for (evaluate-clause c s).

- true: There exists a solution for (cons c f), namely s.
- undef: Choose literal l+ in c such that (evaluate-literal l+ s) returns undef. Let assignment s+ be the result of (cons l+ s+). (evaluate-clause c s+) returns true because (evaluate-literal l+ s+) returns true. Consider any clause c1 in f. We know (evaluate-clause c1 s+) returns true, because (evaluate-clause c1 s) returns true. Therefore, f is satisfied by s+ and there exists a solution for (cons c f), namely s+.
- false: Make a new assignment s* such that (evaluate-literal l s*) is true by removing (negate l) from s and then adding l to s. By construction, we have that (evaluate-clause c s*) is true. We now need to show (evaluate-clause c1 s*) is true for all c1 in f.

Consider a clause c1 in f. If literal (negate l) is not a member of c1, then we know that (evaluate-clause c1 s*) is still true (because l is the only literal that changed in s). Recall c has RATp so the resolvent r computed by (resolution l c c1) has ATp with respect to f. By ATp-lemma, r is also satisfied by s. Therfore, there exists a literal

`lr` in `r` such that `(evaluate-literal lr s)` is `true`. Now, `lr` cannot be in `c` because `(evaluate-clause c s)` is `false`. Because `lr` is in `r` and not in `c`, we know `lr` is in `c1`. Furthermore, `lr` cannot be equal to `(negate l)` because `(negate l)` is not in `r` by the definition of resolution. Therefore, `(evaluate-clause c1 s*)` is `true`, and there exists a solution for `(cons c f)`, namely `s*`.

Commentary. One key observation during the proof of `RATp-lemma` was the need for a case split on `(evaluate-clause c s)`. We previously tried to use an induction on clause list from `RATp1` with `(not (truep (evaluate-clause c s)))` as a hypothesis, which was insufficient. Feedback from ACL2 led us to a full three-way case split, which strengthened the condition to `(falsep (evaluate-clause c s))`.

Another subtle part of the proof was the case of `tautologyp` for the resolvent during an induction of the clause list in `RATp1`. In this proof, we needed to find a conflicting literal in the resolvent and then show that the existence of a conflicting literal implies that a clause from the formula is satisfied by the modified solution.

4.3 Statistics

Our RAT proof checker formalization, specification, and mechanical proof of correctness[3] contain 93 ACL2 `defun` events and 282 ACL2 `defthm` events and certifies in approximately 45 seconds. Of those, 32 `defun` and 140 `defthm` events are specific to the RAT proof checker while the other 61 `defun` and 142 `defthm` events are part of our "library" of SAT concepts. This library includes code about sets, literals, clauses, evaluation, unit propagation, and parsing. The RAT proof checker contains 1088 lines of uncommented non-blank lines of ACL2 source and 2080 lines total; the associated SAT library contains 1121 uncommented non-blank lines of ACL2 and 1836 lines total. We can extract only the definitions that are necessary to create an executable version of the RAT proof checker; this can be expressed in just 26 `defun` events. ACL2 allows the user to supply custom hints for conjectures that are not proved automatically; hints are used to guide the theorem prover towards a proof. We had to provide custom hints for 33 of the 140 `defthm` events that are specific to the RAT proof checker and 12 of the 142 `defthm` events in the SAT library.

5 Conclusion

We presented the formalization, specification, and proof of correctness for a SAT proof checker in ACL2. Our proof checker is based the strong redundancy property RAT that preserves satisfiability. This allows us to validate refutations generated by state-of-the-art SAT solvers that make use of techniques based on extended resolution. We describe the first mechanically-verified proof checker to be complete with respect to all contemporary SAT-solving techniques.

We are developing a faster checker that will employ watched-literal data structures. Our current checker, although proven to be correct, is still too slow to use to check large

[3] The implementation and proof presented in this paper are available at the address `http://cs.utexas.edu/~nwetzler/itp13/`

proofs. We do not expect it to be too difficult to map clauses from list data structures to memory arrays represented in a heap since the ACL2 formalization of arrays is actually given with a list-based semantics. The inclusion of pointers to implement watched-literals will require that we prove an invariant assuring that the watched-literal data structure is always properly maintained. A fast, verified RAT proof checker could be used improve ACL2 by way of a verified clause processor. In other words, ACL2 could construct a SAT encoding for a given subgoal and then call any off-the-shelf SAT solver that produces a solution or a proof trace, that could then be checked by our tool to admit a theorem of unsatisfiablity into the logic. A framework for this proof strategy is explored by Davis et al. in [33].

We submit that all SAT solvers should be able to emit UNSAT proofs that can be checked. Experimentation has shown us that UNSAT proofs can generally be checked using a C-based program with watched literals in a time similar to that required by contemporary solvers. In the future, we encourage all SAT-solver developers to make provisions for emitting RAT proof traces that can be verified using a checker, like the one presented here. Furthermore, it should be the focus of future research to devise elegant and efficient ways of producing RAT proof traces.

References

1. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 114–121. IEEE (2001)
2. Lescuyer, S., Conchon, S.: A reflexive formalization of a SAT solver in Coq. In: International Conference on Theorem Proving in Higher Order Logics, TPHOLs (2008)
3. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM (JACM) 7(3), 201–215 (1960)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
6. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. Electronic Notes in Theoretical Computer Science 269, 3–17 (2011)
7. Marić, F.: Formalization and implementation of modern SAT solvers. Journal of Automated Reasoning 43(1), 81–119 (2009)
8. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science 411(50), 4333–4356 (2010)
9. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-Driven Clause Learning SAT Solvers. Handbook of Satisfiability, ch. 4, pp. 131–153. IOS Press (February 2009)
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
11. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 363–378. Springer, Heidelberg (2012)
12. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 880–885. IEEE (2003)

13. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics, ISAIM (2008)
14. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation 4(75-97), 45 (2008)
15. Weber, T.: Efficiently checking propositional resolution proofs in Isabelle/HOL. In: International Workshop on the Implementation of Logics (IWIL), vol. 212, pp. 44–62 (2006)
16. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2009)
17. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 260–274. Springer, Heidelberg (2010)
18. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Wener, B.: Verifying SAT and SMT in Coq for a fully automated decision procedure. In: PSATTT 2011: International Workshop on Proof-Search in Axiomatic Theories and Type Theories (2011)
19. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 10886–10891. IEEE (2003)
20. Oe, D., Stump, A.: Combining a logical framework with an RUP checker for SMT proofs. In: Satisfiability Modulo Theories (SMT), p. 40 (2011)
21. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Proceedings of Haifa Verification Conference (to appear, 2013)
22. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics 96-97, 149–176 (1999)
23. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Fox, M., Poole, D. (eds.) Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI). AAAI Press (2010)
24. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning 2, pp. 466–483. Springer (1983)
25. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
26. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Verifying refutations with extended resolution. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 345–359. Springer, Heidelberg (2013)
27. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston (2000)
28. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research (JAIR) 22, 319–351 (2004)
29. Haken, A.: The intractability of resolution. Theoretical Computer Science 39, 297–308 (1985)
30. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4), 28–32 (1976)
31. Marques Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
32. Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
33. Davis, J., Swords, S.: Verified AIG Algorithms in ACL2. In: Gamboa, R., Davis, J. (eds.) Proceedings International Workshop on the ACL2 Theorem Prover and its Applications, ACL2. EPTCS, vol. 114, pp. 95–110 (2013), http://dx.doi.org/10.4204/EPTCS.114

Formalizing Bounded Increase*

René Thiemann

Institute of Computer Science, University of Innsbruck, Austria
rene.thiemann@uibk.ac.at

Abstract. Bounded increase is a termination technique where it is tried to find an argument x of a recursive function that is increased repeatedly until it reaches a bound b, which might be ensured by a condition $x < b$. Since the predicates like $<$ may be arbitrary user-defined recursive functions, an induction calculus is utilized to prove conditional constraints.

In this paper, we present a full formalization of bounded increase in the theorem prover Isabelle/HOL. It fills one large gap in the pen-and-paper proof, and it includes generalized inference rules for the induction calculus as well as variants of the Babylonian algorithm to compute square roots. These algorithms were required to write executable functions which can certify untrusted termination proofs from termination tools that make use of bounded increase. And indeed, the resulting certifier was already useful: it detected an implementation error that remained undetected since 2007.

1 Introduction

A standard approach to proving termination of recursive programs is to find a well-founded order $\succ$, such that every recursive call *decreases* w.r.t. this order: whenever $f(\ell_1, \ldots, \ell_n) = C[f(r_1, \ldots, r_n)]$ is a defining equation for f, then $(\ell_1, \ldots, \ell_n) \succ (r_1, \ldots, r_n)$ has to hold. For example, one can use a measure function m which maps values into the naturals where $(\ell_1, \ldots, \ell_n) \succ (r_1, \ldots, r_n)$ is defined as $m(\ell_1, \ldots, \ell_n) > m(r_1, \ldots, r_n)$. For this approach to be sound, one uses well-foundedness of the $>$-order on the naturals.

However, often termination can also be concluded since some argument is *increased* until it exceeds a bound, as demonstrated in the following algorithm which computes $\sum_{i=0}^{n} i$.

$$\begin{aligned} &\mathit{compute\text{-}sum}\ n = \mathit{sum}\ 0\ n \\ &\qquad \mathit{sum}\ i\ n = \mathsf{if}\ i \leq n\ \mathsf{then}\ i + \mathit{sum}\ (i+1)\ n\ \mathsf{else}\ 0 \end{aligned}$$

Also here an appropriate measure can be chosen which is decreased in every iteration, e.g. the difference of the parameters: $m(i, n) = n - i$. However, one needs more information to conclude termination: the measure maps into the integers (and not into the naturals), where the $>$-order is not well-founded.

* Supported by the Austrian Science Fund (FWF), project P22767.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 245–260, 2013.

Only in combination with the condition $i \leq n$ one can conclude that the value $n - i$ is bounded from below and thus, termination of the function.[1]

In the remainder of the paper, we are focussing on term rewrite systems (TRSs), a simple yet powerful computational model that underlies much of declarative programming and theorem proving—for example, both Haskell-functions and Isabelle/HOL-functions can be translated into TRSs such that termination of the TRSs implies termination of the functions [7,15]. For TRSs, the above kind of termination argument has been introduced as the termination technique *bounded increase* [9], where measure functions like $m(i, n) = n - i$ are allowed in the form of polynomial orders [16].

To improve the reliability of automated termination tools for TRSs like AProVE [8] and T$_{\text{T}}$T$_2$ [13], we already formalized several termination techniques (IsaFoR) which resulted in a certifier (CeTA) which validates or invalidates untrusted termination proofs [20]. Other certifiers are based on the alternative formalizations Coccinelle [6] and CoLoR [5]. Due to the certifiers, bugs have been detected in the automated termination tools, which remained without notice for several years and could easily be fixed after their detection.

However, CeTA can only certify those proofs where all applied termination techniques have been formalized. And in order to achieve a high coverage, the integration of bounded increase is essential. For example, in the latest international termination competition in 2012, two versions of AProVE participated: one unrestricted version, and one which only uses techniques that are supported by CeTA; a detailed inspection of the proofs revealed that for more than half of the termination proofs where only the unrestricted version was successful, the method of bounded increase was applied.

Although the termination argument behind bounded increase looks quite intuitive, we want to stress that the underlying soundness proofs are far from being trivial. To this end, consider the following TRS which is a reformulation of the previous algorithm as TRS. We list three problems that are solved in [9].

$$\begin{aligned}
\mathsf{compute\text{-}sum}(n) &\to \mathsf{sum}(0, n)\\
\mathsf{sum}(i, n) &\to \mathsf{if}_{\mathsf{sum}}(i \leq n, i, n)\\
\mathsf{if}_{\mathsf{sum}}(\mathsf{true}, i, n) &\to i + \mathsf{sum}(i + 1, n)\\
\mathsf{if}_{\mathsf{sum}}(\mathsf{false}, i, n) &\to 0
\end{aligned}$$

1. Since interpretations are used which interpreted into the integers, for strong normalization one needs to know that there is a bound on the integers, which is obtained from analyzing side conditions.
2. Usually, for termination analysis of TRSs one uses orders which are weakly monotone w.r.t. contexts, i.e., whenever $\ell \gtrsim r$ then $C[\ell] \gtrsim C[r]$. However, polynomials like $n - i$ result in non-monotone orders $\gtrsim$.

[1] Alternatively, one can define $m(i, n) = n + 1 \dot{-} i$ which maps into the naturals using the truncating subtraction $\dot{-}$. But again one requires the condition $i \leq n$ for the termination proof, namely to ensure $m(i, n) > m(i + 1, n)$.

3. Since there is no builtin arithmetic for pure term rewriting, one adds rewrite rules which compute $\leq$, $+$, etc., like the following ones in $\mathcal{R}_{\mathsf{arith}}$:

$$\begin{aligned} 0 \leq y &\to \mathsf{true} & 0 + y &\to y \\ \mathsf{s}(x) \leq 0 &\to \mathsf{false} & \mathsf{s}(x) + y &\to \mathsf{s}(x + y) \\ \mathsf{s}(x) \leq \mathsf{s}(y) &\to x \leq y \end{aligned}$$

Concrete numbers n in $\mathcal{R}_{\mathsf{sum}}$ are directly replaced by $\mathsf{s}^n(0)$ where s and 0 are the constructors for natural numbers.

As a consequence, conditions like $i \leq n$ are not arithmetic constraints, but rewrite constraints $i \leq n \to^*_{\mathcal{R}} \mathsf{true}$ where $\mathcal{R}$ is a TRS which contains the rules of $\mathcal{R}_{\mathsf{arith}}$. As a result, one has to be able to solve conditional constraints of the form $t_1 \to^*_{\mathcal{R}} t_2 \longrightarrow t_3 \succ t_4$.

The paper is structured as follows: after the preliminaries in Sect. 2, we shortly recapitulate the solutions in [9] to all three problems in Sect. 3–5 and present their formalization. Here, major gaps in the original proofs are revealed, and the existing results are generalized. In Sect. 6 we illustrate complications that occurred when trying to extend our certifier towards bounded increase, which also forced us to formalize a precise algorithm to compute square roots.

2 Preliminaries

We briefly recall some basic notions of term rewriting [2].

The set of (first-order) terms over some signature $\mathcal{F}$ and variables $\mathcal{V}$ is written as $\mathcal{T}(\mathcal{F}, \mathcal{V})$. A context C is a term with one hole $\Box$, and $C[t]$ is the term where $\Box$ is replaced by t. The term t is a subterm of s iff $s = C[t]$ for some C, and it is a proper subterm if additionally $s \neq t$. A substitution σ is a mapping from variables to terms. It is extended homomorphically to terms where we write $t\sigma$ for the application of σ on t. A TRS is a set of rules $\ell \to r$ for terms ℓ and r. The rewrite relation of a TRS $\mathcal{R}$ is defined as $s \to_{\mathcal{R}} t$ iff $s = C[\ell\sigma]$ and $t = C[r\sigma]$ for some $\ell \to r \in \mathcal{R}$, σ, and C. By $NF(\mathcal{R})$ we denote the normal forms of $\mathcal{R}$, i.e., those terms s where there is no t such that $s \to_{\mathcal{R}} t$. A substitution σ is *normal w.r.t.* $\mathcal{R}$ iff $\sigma(x) \in NF(\mathcal{R})$ for all x. A symbol f is *defined* w.r.t. $\mathcal{R}$ iff there is some rule $f(\dots) \to r \in \mathcal{R}$. The remaining symbols are called *constructors* of $\mathcal{R}$.

Throughout this paper we are interested in the innermost rewrite relation $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ which is defined like $\to_{\mathcal{R}}$ with the additional requirement that all proper subterms of $\ell\sigma$ must be normal forms w.r.t. $\mathcal{R}$.[2] We write $SIN_{\mathcal{R}}(t)$ to indicate

[2] In IsaFoR we consider a more general definition of innermost rewriting, where the proper subterms of $\ell\sigma$ must be normal forms w.r.t. some TRS $\mathcal{Q}$ which is independent from $\mathcal{R}$. All results in this paper have been proven for this generalized notion of rewriting under the condition that $NF(\mathcal{Q}) \subseteq NF(\mathcal{R})$. This generalization was important, as it is also utilized in the termination proofs that are generated by AProVE. Moreover, the generalized innermost rewrite relation has the advantage that it is monotone w.r.t. $\mathcal{R}$. However, to improve the readability, in this paper we just use $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$, i.e., we fix $\mathcal{Q} = \mathcal{R}$.

that the term t is strongly normalizing w.r.t. the innermost rewrite relation, i.e., there is no infinite derivation $t \xrightarrow{\mathsf{i}}_{\mathcal{R}} t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}} t_2 \xrightarrow{\mathsf{i}}_{\mathcal{R}} \ldots$, and $SIN(\mathcal{R})$ denotes innermost termination of $\mathcal{R}$, i.e., strong normalization of $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$.

A popular way to prove innermost termination of TRSs is to use dependency pairs [1,10], which capture all calls of a TRS.

Example 1. Let $\mathcal{R} := \mathcal{R}_{\mathsf{sum}} \cup \mathcal{R}_{\mathsf{arith}}$ where additionally the number 1 in $\mathcal{R}_{\mathsf{sum}}$ has been replaced by $\mathsf{s}(\mathsf{0})$. Then the following set $\mathcal{P}$ contains those dependency pairs of $\mathcal{R}$ which correspond to recursive calls.

$$\mathsf{sum}^\sharp(i,n) \to \mathsf{if}^\sharp_{\mathsf{sum}}(i \leq n, i, n) \quad (1) \qquad \mathsf{s}(x) \leq^\sharp \mathsf{s}(y) \to x \leq^\sharp y \quad (3)$$

$$\mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, i, n) \to \mathsf{sum}^\sharp(i + \mathsf{s}(\mathsf{0}), n) \quad (2) \qquad \mathsf{s}(x) +^\sharp y \to x +^\sharp y \quad (4)$$

Here, the sharped-symbols are fresh *tuple*-symbols ($\mathcal{F}^\sharp$) which can be treated differently from their defined counterparts when parametrizing orders, etc.

The major theorem of dependency pairs tells us for Ex. 1 that $\mathcal{R}$ is innermost terminating iff there is no (minimal) innermost $(\mathcal{P},\mathcal{R})$-chain. Here, a chain is an infinite derivation of the form

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} \ldots \qquad (\star)$$

where all $s_i \to t_i \in \mathcal{P}$ and all $s_i\sigma_i \in NF(\mathcal{R})$. The chain is minimal if $SIN_{\mathcal{R}}(t_i\sigma_i)$ for all i. Intuitively, the step from $s_i\sigma$ to $t_i\sigma$ corresponds to a recursive call, and the step from $t_i\sigma$ to $s_{i+1}\sigma_{i+1}$ evaluates the arguments before the next call.

In the remainder of this paper we assume some fixed TRS $\mathcal{R}$ and in examples we always use the TRS $\mathcal{R}$ of Ex. 1. Consequently, in the following we use the notion of $\mathcal{P}$-chain instead of $(\mathcal{P},\mathcal{R})$-chain where $\mathcal{P}$ is some set of (dependency) pairs $s \to t$. We call $\mathcal{P}$ *terminating* iff there is no minimal $\mathcal{P}$-chain.[3]

One major technique to show termination of $\mathcal{P}$ is the reduction pair processor [10] which removes pairs from $\mathcal{P}$—and termination can eventually be concluded if all pairs have been removed. This technique has been formalized for all certifiers [5,6,20].

Theorem 2. *Let $(\succsim, \succ)$ be a* reduction pair*, i.e., $\succ$ is an order which is strongly normalizing, $\succsim$ is a quasi-order satisfying $\succsim \circ \succ \circ \succsim \subseteq \succ$, both $\succ$ and $\succsim$ are stable (closed under substitutions), and $\succsim$ is monotone (closed under contexts). A set of pairs $\mathcal{P}$ is terminating if $\mathcal{P} \subseteq \succ \cup \succsim$, $\mathcal{R} \subseteq \succsim$, and $\mathcal{P} \setminus \succ$ is terminating.*

One instance of reduction pairs stems from polynomial interpretations over the naturals [16] where every symbol is interpreted by a polynomial over the naturals, and $s \succsim t$ is defined as $[\![s]\!] \geq [\![t]\!]$ (similarly, $s \succ t$ iff $[\![s]\!] > [\![t]\!]$).

[3] In the literature termination of $\mathcal{P}$ is called finiteness of $(\mathcal{P},\mathcal{R})$. We use the notion of termination here, so that "$\mathcal{P}$ is finite" does not have two meanings: finiteness of the set $\mathcal{P}$ or absence of $\mathcal{P}$-chains. In IsaFoR, the latter property is called *finite-dpp*, but as for the innermost rewrite relation, IsaFoR uses a more general notion of chain, which is essential for other termination techniques. To be more precise we do not only consider the 3 components ($\mathcal{P}$, $\mathcal{R}$, and a minimality flag) to define chains, but 7: strict and weak pairs, strict and weak rules, strategy, minimality flag, and a flag to indicate whether substitutions have to return normal forms on free variables.

3 First Problem: No Strong Normalization

Since for bounded increase interpretations into the integers are used where $>$ is not strongly normalizing, the reduction pair processor has to be adapted, as the resulting strict order $\succ$ is not strongly normalizing [9]. Instead, it is required that $\succ$ is *non-infinitesimal*, which is defined as absence of infinite sequences $t_1 \succ t_2 \succ \ldots$ which additionally satisfy $t_i \succ \mathsf{c}$ for all i where c is some constant. However, weakening strong normalization to being non-infinitesimal requires to strengthen other preconditions in the reduction pair processor: in detail, termination of $\mathcal{P}\backslash\succ$ does no longer suffice, but additionally one requires that termination is ensured if all bounded pairs are removed, i.e., termination of $\mathcal{P} \setminus \{s \to t \mid s \succsim \mathsf{c}\}$.

Another adaptation is required when trying to solve term constraints, which is a distinction between symbols of $\mathcal{F}$ and tuple symbols. It is easily motivated: if every symbol is mapped to an integer, then polynomial constraints like $2x \geq x$ are no longer satisfied, since x might be instantiated by some negative value. However, $2x \geq x$ holds over the naturals. Since one wants to be able to conclude $2x \geq x$, one uses interpretations where all $\mathcal{F}$-symbols are interpreted by naturals, and only the tuple symbols in $\mathcal{F}^\sharp$ may interpret into the integers. As a consequence, $2x \geq x$ can be concluded, but $\succ$ and $\succsim$ are no longer stable, since we only get closure under those substitutions which instantiate all variables by terms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$—this property is called $\mathcal{F}$-stable and these substitutions are called $\mathcal{F}$-substitutions.

Of course, if one wants to use $\mathcal{F}$-stable orders $\succsim$ and $\succ$ in the reduction pair processor, one has to know that all substitutions σ_i in a chain ($\star$) can be chosen as $\mathcal{F}$-substitutions, a property which is named signature extension: if we can prove termination if we only consider $\mathcal{F}$-substitutions, then we can prove termination for arbitrary larger signatures like $\mathcal{F} \cup \mathcal{F}^\sharp$. In [9, proof of Thm. 11, technical report] signature extensions are taken for granted ("clearly"), most likely since signature extensions are possible for (innermost) termination analysis on the level of TRSs. However, in [17] we proved that signature extensions are unsound on the level of minimal chains. Luckily, we were able to formalize that for (minimal) innermost chains, signature extensions are indeed sound. The efforts to get this result is in stark contrast to "clearly": $\approx$ 900 lines, cf. the locale `cleaning-innermost` within the theory `Signature-Extension.thy`.

Concerning the restriction to $\mathcal{F}$-stable orders, in the formalization we also added a new feature which is not present in [9]: the restriction to $\mathcal{F}$-monotone orders. For example, if we interpret $[\![f]\!](x) = x^2$, then the resulting order $\succsim$ is not monotone, as the arguments range over all integers. For $\mathcal{F}$-monotone orders it is instead only required that these are monotone if all arguments are terms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$, i.e., in the case of polynomial orders, if the arguments evaluate to natural numbers. Hence, the interpretation $[\![f]\!](x) = x^2$ is $\mathcal{F}$-monotone.

Since we need to prove termination of $\mathcal{P} \setminus \{s \to t \mid s \succsim \mathsf{c}\}$, it is essential that at least one of the pairs is bounded. However, if we use an interpretation like $[\![\mathsf{sum}^\sharp]\!](i, n) = [\![\mathsf{if}^\sharp_{\mathsf{sum}}]\!](b, i, n) = n - i$, then none of the pairs (1) and (2) is bounded. So, we will not make progress using this interpretation without further adaptations. To this end, conditions are added where for every pair in a chain, one

obtains the conditions from the adjacent pairs in the chain: for example, when considering the preceding rewrite steps of some pair $s \to t$, instead of demanding $s \gtrsim \mathsf{c}$ for boundedness, one can demand that all implications $v \xrightarrow{\mathsf{i}}{}^* s \longrightarrow s \gtrsim \mathsf{c}$ are satisfied for every variable renamed pair $u \to v \in \mathcal{P}$ and every instantiation of the variables with normal forms.

More formally, we consider conditional constraints which are first-order formulas using atomic formulas of the form $s \xrightarrow{\mathsf{i}}{}^* t$, $s \gtrsim t$, and $s \succ t$. The satisfaction relation for normal $\mathcal{F}$-substitutions σ (written $\sigma \models \phi$) is defined as follows.

$$\sigma \models \phi \text{ iff } \begin{cases} s\sigma \gtrsim t\sigma, & \text{if } \phi = s \gtrsim t \\ s\sigma \succ t\sigma, & \text{if } \phi = s \succ t \\ s\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t\sigma \wedge NF(t\sigma) \wedge SIN(s\sigma), & \text{if } \phi = s \xrightarrow{\mathsf{i}}{}^* t \end{cases}$$

The other logic connectives $\longrightarrow$, $\wedge$, $\forall$ are treated as usual, and $\models \phi$ is defined as $\sigma \models \phi$ for all normal $\mathcal{F}$-substitutions σ. Using these definitions, a constraint like $s \gtrsim \mathsf{c}$ can be replaced by $\bigwedge u\ v.\ u \to v \in \mathcal{P} \implies \models v \xrightarrow{\mathsf{i}}{}^* s \longrightarrow s \gtrsim \mathsf{c}$.

The most tedious part when integrating this adaptation of conditional constraints into the formalization of the reduction pair processor was the treatment of variable renamed pairs: in ($\star$) we use several different substitutions of non-renamed pairs, and we had to show that instead one can use one substitution where all pairs are variable renamed. And of course, when certifying termination proofs, one again has to work modulo variable names, since one does not know in advance how the variables have been renamed apart in the termination tool.

4 Second Problem: No Monotonicity

Monotonicity and stability are important to ensure the implication $\mathcal{R} \subseteq \gtrsim \implies \xrightarrow{\mathsf{i}}_{\mathcal{R}} \subseteq \gtrsim$. Using this implication with the requirement $\mathcal{R} \subseteq \gtrsim$ in the reduction pair processor allows to replace all $\xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}}$ in ($\star$) by $\gtrsim$.

For innermost rewriting, the requirement that all rules have to be *weakly decreasing* ($\mathcal{R} \subseteq \gtrsim$ or equivalently, $\ell \gtrsim r$ for all $\ell \to r \in \mathcal{R}$) can be reduced to demand a weak decrease of only the *usable rules*. Here, the usable rules of a term t are those rules of $\mathcal{R}$ which can be applied when reducing $t\sigma$ at those positions which are relevant w.r.t. the order $\gtrsim$; and the usable rules of $\mathcal{P}$ are the usable rules of all right-hand sides of $\mathcal{P}$. We omit a formal definition (e.g., [10, Def. 21]) here, but just illustrate the usage in our running example.

Example 3. We use the reduction pair processor (without the adaptations from the previous section) in combination with the polynomial order $[\![\mathsf{sum}^\sharp]\!](i, n) = [\![\mathsf{if}^\sharp_{\mathsf{sum}}]\!](b, i, n) = 0$ and $[\![+^\sharp]\!](x, y) = [\![\leq^\sharp]\!](x, y) = [\![\mathsf{s}]\!](x) = 1 + x$.

The only rules that are required to evaluate the arguments of right-hand sides of $\mathcal{P} = \{(1)\text{–}(4)\}$ are the $\leq$-rules for (1) and the $+$-rules for (2). However, since the first arguments of $\mathsf{sum}^\sharp$ and $\mathsf{if}^\sharp_{\mathsf{sum}}$ are ignored by the polynomial order, there are no usable rules. Since moreover, all pairs are at least weakly decreasing, we can delete the strictly decreasing pairs (3) and (4) by the reduction pair processor. Hence, it remains to prove termination of $\mathcal{P} \setminus \succ = \{(1), (2)\}$.

From the soundness result for usable rules [10, Lem. 23] one derives that whenever $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s$ for some normal substitution σ and the usable rules of t are weakly decreasing, then $t\sigma \succsim s$. Unfortunately, this lemma does no longer hold if $\succsim$ is not monotone. To solve this problem, in [9] a monotonicity function $ord :: \mathcal{F} \cup \mathcal{F}^\sharp \Rightarrow \mathbb{N} \Rightarrow \{0, 1, -1, 2\}$[4] is defined. It determines for each symbol f and each argument i, which relationship between s_i and t_i must be ensured to guarantee $C[s_i] \succsim C[t_i]$ for some context $C = f(u_1, \ldots, u_{i-1}, \Box, u_{i+1}, \ldots, u_n)$:

- if there are no requirements on s_i and t_i then $ord(f, i)$ is defined as 0,
- if $s_i \succsim t_i$ is required then $\succsim$ is monotonically increasing in the i-th argument and $ord(f, i)$ is defined as 1,
- if $t_i \succsim s_i$ is required then $\succsim$ is monotonically decreasing in the i-th argument and $ord(f, i)$ is defined as -1,
- otherwise, $ord(f, i)$ is defined as 2 (and it is demanded that $s_i \succsim t_i \wedge t_i \succsim s_i$ suffices to ensure $C[s_i] \succsim C[t_i]$).

Using *ord* the generalized usable rules are defined as follows.

Definition 4. *Let $\mathcal{R}$ be some TRS. The generalized usable rules of term t are defined as the least set $\mathcal{U}(t)$ such that*

- *whenever $f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}$, then $f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{U}(f(t_1, \ldots, t_n))$,*
- *whenever $\ell \to r \in \mathcal{U}(t)$ then $\mathcal{U}(r) \subseteq \mathcal{U}(t)$, and*
- *$\mathcal{U}(t_i)^{ord(f,i)} \subseteq \mathcal{U}(f(t_1, \ldots, t_n))$.*

Here, $U^0 = \emptyset$, $U^1 = U$, $U^{-1} = \{r \to \ell \mid \ell \to r \in U\}$, and $U^2 = U \cup U^{-1}$.
The generalized usable rules of a set of pairs $\mathcal{P}$ is $\mathcal{U}(\mathcal{P}) = \bigcup_{s \to t \in \mathcal{P}} \mathcal{U}(t)$.

Example 5. In Ex. 3 we remained with the pairs $\{(1), (2)\}$. If we choose the polynomial order defined by $[\![\mathsf{sum}^\sharp]\!](i, n) = [\![\mathsf{if}^\sharp_{\mathsf{sum}}]\!](b, i, n) = n - i$, $[\![+]\!](x, y) = x + y$, $[\![\mathsf{0}]\!] = 0$, and $[\![\mathsf{s}]\!](x) = 1 + x$, then $\mathcal{U}(\{(1), (2)\}) = \mathcal{U}(\mathsf{if}^\sharp_{\mathsf{sum}}(i \le n, i, n)) \cup \mathcal{U}(\mathsf{sum}^\sharp(i{+}\mathsf{s}(\mathsf{0}), n))$ are the two reversed $+$-rules $y \to \mathsf{0}{+}y$ and $\mathsf{s}(x{+}y) \to \mathsf{s}(x){+}y$ which are both decreasing by this polynomial order, i.e., $\mathcal{U}(\{(1), (2)\}) \subseteq \succsim$.

The results on the generalized usable rules and usable rules are similar.

Lemma 6. *Whenever $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s$ for some normal substitution σ, then $\mathcal{U}(t) \subseteq \succsim$ implies $t\sigma \succsim s$.*

Concerning the formalization of this result, we only added one indirection: Instead of directly defining $\mathcal{U}$ using *ord*, we define $\mathcal{U}_\pi$ like $\mathcal{U}$ where *ord* is replaced by some user defined function $\pi :: \mathcal{F} \cup \mathcal{F}^\sharp \Rightarrow \mathbb{N} \Rightarrow \{0, 1, -1, 2\}$ and then demand that π and $\succsim$ are *compatible*: Whenever $\pi(f, i) = k$ and $1 \le i \le \mathrm{arity}(f)$, then $\succsim$ must satisfy the monotonicity condition which is required for $ord(f, i) = k$.

The reason is that usually, we can only approximate $ord(f, i)$, but not decide it. For example, for a polynomial interpretation over the naturals with $[\![f]\!](x) = x^2 - x$ we have $ord(f, 1) = 1$, since $x^2 - x$ is monotonically increasing over

[4] In IsaFoR, we use a datatype to indicate the monotonicity instead of $\{0, 1, -1, 2\}$. It consists of the elements `Ignore` (0), `Increase` (1), `Decrease` (-1), and `Wild` (2).

the naturals. However, we also allow to define $\pi(f,1) = 2 \neq ord(f,i)$ with the consequences, that there are more usable rules, but weaker requirements for the monotonicity check.

Using this adaptation we proved Lem. 6 where $\mathcal{U}$ is replaced by $\mathcal{U}_\pi$ and where compatibility of π and $\succsim$ is added as additional assumption. Even with these changes, we could show the result by minor adaptations of the original proof.

However, it turns out that Lem. 6 is not really helpful in the overall approach of bounded increase as it is assumes that $\succsim$ is stable. But from the last section we know that we are more interested in relations $\succsim$ which are only $\mathcal{F}$-stable. And then Lem. 6 does no longer hold, even if we restrict σ to be an $\mathcal{F}$-substitution. To still achieve a result similar to Lem. 6 for $\mathcal{F}$-stable $\succsim$, we could additionally demand that $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$. Although this would be sound, it is still not useful, as we require the result for terms t like $\mathsf{sum}^\sharp(i + \mathsf{s}(\mathsf{0}), n)$ where the root symbol is a tuple symbol that is not in $\mathcal{F}$.

Hence, we formalized the following lemma where we also improved the definition of *ord* (or compatibility) by adding ideas from $\mathcal{F}$-monotonicity: all terms s_i, t_i, and u_i that are occurring in the definition of *ord* (or compatibility) are only quantified over $\mathcal{T}(\mathcal{F},\mathcal{V})$. Note that every tuple symbol is a constructor.

Lemma 7. *If $\mathcal{R}$ is a TRS over signature $\mathcal{F}$, π and $\succsim$ are compatible, σ is a normal $\mathcal{F}$-substitution, $t = f(t_1,\ldots,t_n) \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s$, f is a constructor of $\mathcal{R}$, and all $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$, then $\mathcal{U}_\pi(t) \subseteq \succsim \implies t\sigma \succsim s$.*

The proof of this result was an extended version of the proof for Lem. 6 where the following main property was shown by induction on t for arbitrary k and s.

Lemma 8. *If $\mathcal{R}$ is a TRS over signature $\mathcal{F}$, π and $\succsim$ are compatible, σ is a normal substitution, $t\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}} s$, $t\sigma \in \mathcal{T}(\mathcal{F},\mathcal{V})$ or $t\sigma = f(t_1,\ldots,t_n)$ with constructor f of $\mathcal{R}$ and all $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$, then $\mathcal{U}_\pi(t)^k \subseteq \succsim \implies \{(t\sigma, s)\}^k \subseteq \succsim$ and there are u and a normal substitution δ such that $\mathcal{U}_\pi(u)^k \subseteq \mathcal{U}_\pi(t)^k$ and $s = u\delta$.*

Using these results we are ready to present our formalized result of the adapted, conditional general reduction pair processor which combines both modifications—where $\succ$ does not have to be strongly normalizing and $\succsim$ does not have to be monotone.[5]

Theorem 9. *Let c be some constant. Let $\succsim$ and $\succ$ be $\mathcal{F}$-stable orders where $\succsim$ and $\succ$ are compatible, and $\succ$ is non-infinitesimal. Let π be compatible with $\succsim$.*[6] *Let $\mathcal{P}$, $\mathcal{P}_\succ$, $\mathcal{P}_\succsim$, and $\mathcal{P}_{bound}$ be arbitrary sets of pairs. Then $\mathcal{P}$ is terminating if all of the following properties are satisfied.*

- *$\mathcal{P} \subseteq \mathcal{P}_\succ \cup \mathcal{P}_\succsim$ and $\mathcal{U}_\pi(\mathcal{P}) \subseteq \succsim$*
- *$\mathcal{P} \setminus \mathcal{P}_\succ$ and $\mathcal{P} \setminus \mathcal{P}_{bound}$ is terminating*
- *$\mathcal{R}$ is a TRS over signature $\mathcal{F}$*
- *for all $s \to t \in \mathcal{P}_\succ$ and variable renamed $u \to v \in \mathcal{P}$: $\models v \xrightarrow{\mathsf{i}}{}^* s \longrightarrow s \succ t$*
- *for all $s \to t \in \mathcal{P}_\succsim$ and variable renamed $u \to v \in \mathcal{P}$: $\models v \xrightarrow{\mathsf{i}}{}^* s \longrightarrow s \succsim t$*

[5] In IsaFoR, this processor also integrates the generalization below Thm. 11 in [9], where arbitrary adjacent pairs can be chosen instead of only one preceding pair, cf. lemma `conditional-general-reduction-pair-proc` in `Generalized-Usable-Rules.thy`.

[6] In IsaFoR, all these properties are summarized in the locale `non-inf-order`.

- *for all* $s \to t \in \mathcal{P}_{bound}$ *and variable renamed* $u \to v \in \mathcal{P}$*:* $\models v \xrightarrow{\mathsf{i}}{}^* s \longrightarrow s \gtrsim \mathsf{c}$
- *all pairs in* $\mathcal{P}$ *are of the form* $f(s_1, \ldots, s_n) \to g(t_1, \ldots, t_m)$ *where* g *is a constructor of* $\mathcal{R}$ *and* $s_i, t_j \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ *for all* i *and* j
- *the set of all symbols is countable and* $\mathcal{V}$ *is countably infinite.*

Note that Thm. 9 is similar to [9, Thm. 11] with the difference, that for defining *ord* for the generalized rules we only require $\mathcal{F}$-monotonicity.

The last condition on the cardinality of $\mathcal{F}$ and $\mathcal{V}$ is enforced by our theorem on signature extensions. We believe that it can be weakened to $\mathcal{V}$ being infinite, but this would make the proof for signature extensions more tedious.

Even after having proven Thm. 9, there remain two tasks.

First, we actually require an executable function which computes the set of usable rules (which is defined as an inductive set) for some concrete TRS $\mathcal{R}$ and set of pairs $\mathcal{P}$. To this end, we have not been able to use Isabelle's predicate compiler [3], but instead we used an approach as in [18]: We characterized the generalized usable rules via a reflexive transitive closure. And afterwards, we could just invoke a generic algorithm to compute reflexive transitive closures as the one in [19]. More details can be seen in `Generalized-Usable-Rules-Impl.thy`.

The second task is to deal with conditional constraints that arise from the conditional reduction pair processor, which is discussed in the upcoming section.

5 Third Problem: Solving Conditional Constraints

We start by generating conditional constraints for Ex. 5 via Thm. 9.

Example 10. Using the polynomial order from Ex. 5 we already have a weak decrease for the usable rules. Moreover, all pairs are weakly decreasing and (2) is strictly decreasing, even without considering the conditions: $[\![\mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, i, n)]\!] = n - i > n - (i+1) = [\![\mathsf{sum}^\sharp(i + \mathsf{s}(\mathsf{0}), n)]\!]$. To show that (2) is also bounded—so that it can be removed—we must show $\models \phi_1$ and $\models \phi_2$.

$$\mathsf{if}^\sharp_{\mathsf{sum}}(i' \le n', i', n') \xrightarrow{\mathsf{i}}{}^* \mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, i, n) \longrightarrow \mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, i, n) \gtrsim \mathsf{c} \qquad (\phi_1)$$

$$\mathsf{sum}^\sharp(i' + \mathsf{s}(\mathsf{0}), n') \xrightarrow{\mathsf{i}}{}^* \mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, i, n) \longrightarrow \mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, i, n) \gtrsim \mathsf{c} \qquad (\phi_2)$$

To handle conditional constraints like ϕ_1 and ϕ_2, in [9] an induction calculus is developed with rules to transform conditional constraints, where constraints above the line can be transformed into constraints below the line, cf. Fig. 1. Its aim is to turn rewrite conditions $s \xrightarrow{\mathsf{i}}{}^* t$ into conditions involving the order.

Example 11. Using the induction calculus one can transform ϕ_1 and ϕ_2 into ϕ_3 and ϕ_4. Both of these new constraints are satisfied for the polynomial order of Ex. 5 if one chooses $[\![\mathsf{c}]\!] \le 0$, since then ϕ_3 and ϕ_4 evaluate to $v \ge [\![\mathsf{c}]\!]$ and $v - u \ge [\![\mathsf{c}]\!] \longrightarrow (v+1) - (u+1) \ge [\![\mathsf{c}]\!]$, respectively.

$$\mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, \mathsf{0}, v) \gtrsim \mathsf{c} \qquad (\phi_3)$$

$$\mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, u, v) \gtrsim \mathsf{c} \longrightarrow \mathsf{if}^\sharp_{\mathsf{sum}}(\mathsf{true}, \mathsf{s}(u), \mathsf{s}(v)) \gtrsim \mathsf{c} \qquad (\phi_4)$$

In [9] soundness of the induction calculus is proven: whenever ϕ can be transformed into ϕ' using one of the inference rules, then $\models \phi'$ implies $\models \phi$. During

I. **Constructor and Different Function Symbol**

$$\frac{f(p_1,\ldots,p_n) \xrightarrow{\mathrm{i}}{}^* g(q_1,\ldots,q_m) \wedge \varphi \;\longrightarrow\; \psi}{\mathit{TRUE}}$$ if f is a constructor and $f \neq g$

II. **Same Constructors on Both Sides**

$$\frac{f(p_1,\ldots,p_n) \xrightarrow{\mathrm{i}}{}^* f(q_1,\ldots,q_n) \wedge \varphi \;\longrightarrow\; \psi}{p_1 \xrightarrow{\mathrm{i}}{}^* q_1 \wedge \ldots \wedge p_n \xrightarrow{\mathrm{i}}{}^* q_n \wedge \varphi \;\longrightarrow\; \psi}$$ if f is a constructor

III. **Variable in Equation**

$$\frac{x \xrightarrow{\mathrm{i}}{}^* q \wedge \varphi \;\longrightarrow\; \psi}{\varphi\sigma \;\longrightarrow\; \psi\sigma}$$ if $x \in \mathcal{V}$ and $\sigma = [x/q]$

$$\frac{q \xrightarrow{\mathrm{i}}{}^* x \wedge \varphi \;\longrightarrow\; \psi}{\varphi\sigma \;\longrightarrow\; \psi\sigma}$$ if $x \in \mathcal{V}$, q has no defined symbols, $\sigma = [x/q]$

IV. **Delete Conditions**

$$\frac{\varphi_1 \wedge \ldots \wedge \varphi_n \;\longrightarrow\; \psi}{\varphi'_1 \wedge \ldots \wedge \varphi'_m \;\longrightarrow\; \psi}$$ if $\{\varphi'_1,\ldots,\varphi'_m\} \subseteq \{\varphi_1,\ldots,\varphi_n\}$

V. **Induction**

$$\frac{f(x_1,\ldots,x_n) \xrightarrow{\mathrm{i}}{}^* q \wedge \varphi \;\longrightarrow\; \psi}{\bigwedge_{f(\ell_1,\ldots,\ell_n)\to r \in \mathcal{R}} (r \xrightarrow{\mathrm{i}}{}^* q\sigma \wedge \varphi\sigma \wedge \varphi' \;\longrightarrow\; \psi\sigma)}$$ if $f(x_1,\ldots,x_n)$ does not unify with q

where $\sigma = [x_1/\ell_1,\ldots,x_n/\ell_n]$

and $\varphi' =$
- $\forall y_1,\ldots,y_m.\ f(r_1,\ldots,r_n) \xrightarrow{\mathrm{i}}{}^* q\mu \wedge \varphi\mu \longrightarrow \psi\mu$, if
  - r contains the subterm $f(r_1,\ldots,r_n)$,
  - there is no defined symbol in any r_i,
  - $\mu = [x_1/r_1,\ldots,x_n/r_n]$, and
  - $y_1,\ldots,y_m$ are all occurring variables except $\mathcal{V}(r)$
- TRUE, otherwise

VI. **Simplify Condition**

$$\frac{\varphi \wedge (\forall y_1,\ldots,y_m.\ \varphi' \longrightarrow \psi') \;\longrightarrow\; \psi}{\varphi \wedge \psi'\sigma \;\longrightarrow\; \psi}$$ if $\mathit{DOM}(\sigma) \subseteq \{y_1,\ldots,y_m\}$, there is no defined symbol and no tuple symbol in any $\sigma(y_i)$, and $\varphi'\sigma \subseteq \varphi$

VII. **Defined Symbol without Pairwise Different Variables**

$$\frac{f(p_1,\ldots,p_i,\ldots,p_n) \xrightarrow{\mathrm{i}}{}^* q \wedge \varphi \;\longrightarrow\; \psi}{p_i \xrightarrow{\mathrm{i}}{}^* x \wedge f(p_1,\ldots,x,\ldots,p_n) \xrightarrow{\mathrm{i}}{}^* q \wedge \varphi \longrightarrow \psi}$$ if x is fresh and $p_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$

Fig. 1. Corrected and generalized rules of the induction calculus from [9]

our formalization of this result in `Bounded-Increase(-Impl).thy`, we detected five interesting facts.

Fact 1. Some of the side-conditions in the original definition are not required, but are most likely used to derive a strategy which chooses which rules to apply. For example, in both (V) and (VII) there have been additional side-conditions in [9]. To be more precise, [9] demands that f is defined in (V). Consequently, formulas with a precondition like $f(x) \xrightarrow{\mathsf{i}}{}^* x$ for some constructor f can only be dropped using our version of (V), but not by any of the original rules in [9].

Fact 2. Rule (VII) is unsound in [9] where we had to add the new side-condition $p_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Fact 3. The textual explanation in [9] for the soundness of the induction rule (V) is completely misleading: it is stated that induction can be performed using the length of the reduction $f(x_1, \ldots, x_n)\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} q\sigma$ which is "obviously longer" than the reduction $f(r_1, \ldots, r_n)\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} q\sigma$; only in a footnote it is clarified, that for the actual proof one uses the induction relation $\xrightarrow{\mathsf{i}}_{\mathcal{R}} \circ \trianglerighteq$ which is restricted to innermost terminating terms. This restriction actually is the only reason why minimal chains are considered and why in the semantics of $\sigma \models s \xrightarrow{\mathsf{i}}{}^* t$ one demands $SIN(s\sigma)$.

Note that if one indeed were able to perform induction on the length of the reduction, then one could drop the condition $SIN(s\sigma)$ and consider arbitrary, non-minimal chains. This would have the immediate advantage that bounded increase would be applicable on sets of pairs and TRSs which arise from transformations like [15], where termination of each $t_i\sigma$ in $(\star)$ cannot be ensured, cf. [15, Footnote 3].

The bad news is that we were able to find a counterexample proving that an induction over the length in combination with non-minimal chains is unsound. In detail, consider $\mathcal{R} = \{\mathsf{f} \to \mathsf{g(f)}, \mathsf{f} \to \mathsf{b}, \mathsf{g}(x) \to \mathsf{a}\}$ and $\mathcal{P} = \{\mathsf{h^\sharp(a)} \to \mathsf{h^\sharp(f)}\}$. Then there is a non-minimal innermost chain $\mathsf{h^\sharp(a)} \to_{\mathcal{P}} \mathsf{h^\sharp(f)} \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{h^\sharp(g(f))} \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{h^\sharp(g(b))} \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{h^\sharp(a)} \to_{\mathcal{P}} \ldots$, so we should not be able to prove termination of $\mathcal{P}$ if minimality is dropped from the definition of termination.

However, we show that using the induction calculus—where now $\sigma \models s \xrightarrow{\mathsf{i}}{}^* t$ is defined as $s\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t\sigma \wedge t\sigma \in NF(\mathcal{R})$—in combination with the conditional reduction pair processor we actually can derive termination of $\mathcal{P}$, where we will see that the problem is the induction rule. To this end, we let $\mathsf{h^\sharp(f)} \xrightarrow{\mathsf{i}}{}^* \mathsf{h^\sharp(a)} \longrightarrow \phi$ be some arbitrary conditional constraint that may arise from the generalized reduction pair processor where ϕ might be $\mathsf{h^\sharp(a)} \succ \mathsf{h^\sharp(f)}$ or $\mathsf{h^\sharp(a)} \succsim \mathsf{c}$. Using (II) this constraint is simplified to $\mathsf{f} \xrightarrow{\mathsf{i}}{}^* \mathsf{a} \longrightarrow \phi$. Next, we apply (V), leading to $\mathsf{g(f)} \xrightarrow{\mathsf{i}}{}^* \mathsf{a} \wedge (\mathsf{f} \xrightarrow{\mathsf{i}}{}^* \mathsf{a} \longrightarrow \phi) \longrightarrow \phi$ and $\mathsf{b} \xrightarrow{\mathsf{i}}{}^* \mathsf{a} \longrightarrow \phi$ where the latter constraint is immediately solved by (I). And the former constraint is always satisfied, since $\mathsf{f}\sigma = \mathsf{f} \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} \mathsf{a} = \mathsf{a}\sigma \in NF(\mathcal{R})$ for every $\mathcal{F}$-normal substitution σ.

Fact 4. The main reason for the unsoundness of the induction rule in the previous counterexample is the property that $\mathcal{R}$ is not uniquely innermost normalizing (UIN): f can be evaluated to the two different normal forms a and b.

And indeed, we could show that instead of minimality of the chain, one can alternatively demand UIN of $\mathcal{R}$ to ensure soundness of the induction rule. To be more precise, using UIN of $\mathcal{R}$ we were able to prove soundness of the induction rule via an induction on the distance of $f(x_1, \ldots, x_n)\sigma$ to some normal form.

To have a decidable criterion for ensuring UIN, we further proved that UIN is ensured by confluence of $\mathcal{R}$, and that confluence is ensured by weak orthogonality, i.e., for left-linear $\mathcal{R}$ that only have trivial critical pairs. Here, especially for the latter implication, we required several auxiliary lemmas like the parallel moves lemma, cf. [2, Chapter 6.4] and `Orthogonality.thy`.

Fact 5. As we want to apply Isabelle's code generator [11] on our certification algorithm, we have to formalize conditional constraints via a deep embedding. Therefore, we had the choice between at least two alternatives how to deal with bound variables: we can use a dedicated approach like Nominal Isabelle [21,22], or we perform renamings, α-equivalence, ... on our own.[7]

We first formalized everything using the latter alternative, where we just defined an inductive datatype [4] with constructors for atomic constraints, one constructor for building implications of the shape $\cdot \wedge \cdots \wedge \cdot \longrightarrow \cdot$ (where the set of premises is conveniently represented as a list), and one constructor for universal quantification which takes a variable and a constraint as argument.

Afterwards, we just had to deal with manual renamings at exactly one place: in the definition of applying a substitution on a constraint. Here, for quantified constraints we define $(\forall x.\phi)\sigma = \forall y.(\phi(\sigma\{x := y\}))$ where y is a fresh variable w.r.t. the free variables in σ and ϕ. Note that α-equivalence of two constraints ϕ and ψ can then also easily be checked by just applying the empty substitution on both ϕ and ψ, since the substitution algorithm returns the same result on α-equivalent constraints. Further note, that in our application—certification of transformations on constraints—we require subsumptions checks instead of α-equivalence. Therefore, being able to use equality instead of α-equivalence does not help that much. So, by using an inductive datatype we had minimal effort to integrate renamings. Furthermore, all required algorithms could easily be formulated using the comfort of Isabelle's function package [14].

We additionally tried to perform the same formalization using Nominal Isabelle, since we have been curious what we would gain by using a dedicated package to treat bound variables. Here, our initial attempts have been quite disappointing for the following reasons.

We could not define the datatype for conditional constraints as before, since for the implications we had to manually define two datatypes: one for constraints and one for lists of constraints. Moreover, for several functions which have been accepted without problems using the function package, their nominal counterparts require additional properties which had to be manually proven, including

[7] Of course, also for building sequences of pairs $\mathcal{P}$ (sets of rules) we could have used nominal to avoid manual renamings in rewrite rules. However, since IsaFoR is quite large (over 100,000 lines) we did not want to change the representation of rules until there are extremely good reasons.

new termination proofs. The overall overhead of these manual proofs was in our case by far larger than the one for manual renaming. Moreover, as far as we know, also for code generation some manual steps would be necessary. In the end, we aborted our attempt to have a fully formalized version of conditional constraints which are based on Nominal Isabelle.

6 Babylonian Square Root Algorithms

Eventually, all required theorems for bounded increase have been formalized, and certification algorithms have been integrated in CeTA. Clearly, we tried to certify the bounded increase proofs that have been generated by AProVE. Here, nearly all proofs have been accepted, except for two problematic kinds of proofs.

The one problem was that CeTA discovered a real implementation bug which remained undetected since 2007: AProVE applies (IV) also inside induction hypotheses, i.e., under certain circumstances it simplified $(\psi \longrightarrow \phi) \longrightarrow \phi$ to $(\emptyset \longrightarrow \phi) \longrightarrow \phi$ as $\emptyset \subseteq \{\psi\}$. After the bug has been fixed, for one of the TRSs, a termination proof could not be generated anymore.

The other problem has been an alternative criterion to ensure boundedness, cf. the following example.

Example 12. For the TRS GTSSK07/cade14 from the termination problem database, the generalized reduction pair processor is used with the interpretation where $[\mathsf{diff}^\sharp](x_1, x_2) = -1+x_1^2+x_2^2-2x_1x_2$, $[\mathsf{s}](x) = x+1$, $[\mathsf{0}] = 0$, and $[\mathsf{c}] = -1$.[8] After applying the induction calculus, one of the constraints is $\mathsf{diff}^\sharp(\mathsf{s}(\mathsf{0}), x) \gtrsim \mathsf{c}$ which is equivalent to $x^2 - 2x \geq -1$. This inequality is valid, but it cannot be shown using the standard criterion of absolute positiveness [12], which is also the criterion that is used in CeTA. Here, the trick is, that $[\mathsf{diff}^\sharp](x_1, x_2) = (x_1-x_2)^2-1$ and hence $[\mathsf{diff}^\sharp](\dots) \geq 0 - 1 \geq -1 = [\mathsf{c}]$.

In fact, in the previous example AProVE was configured such that tuple symbols are interpreted as $[f^\sharp](x_1, \dots, x_n) = f_0 + (f_1x_1 + \dots + f_nx_n)^2$ for suitable values f_i. Then $[\mathsf{c}]$ is chosen as $\min_{f^\sharp \in \Sigma} f_0$ such that by construction, $f^\sharp(\dots) \gtrsim \mathsf{c}$ always holds and no constraints have to be checked for boundedness.

If we knew that the tuple symbols are interpreted in this way and all values f_i were provided, it would be easy to conclude boundedness. However, we refrained from adding a special format to express interpretations of this shape, as it would require a new dedicated pretty printer in AProVE. Instead, we want to be able to detect during certification, whether for some arbitrary interpretation $[f^\sharp](x_1, \dots, x_n) = p$ we can find values f_i such that p is equivalent to $f_0 + (f_1x_1 + \dots + f_nx_n)^2$ (which is equivalent to $f_0 + \sum f_i^2x_i^2 + \sum 2f_if_jx_ix_j$).

To this end, we first transform p into summation normal form $a_0+\sum a_ix_i^2+\dots$ with concrete values a_i, and afterwards we figure out all possible values for each f_i: $f_0 = a_0$ and $f_i = \pm\sqrt{a_i}$ for $i > 0$. For each possible combination of $(f_0, \dots, f_n)$ we can then just check whether $p = f_0 + (f_1x_1 + \dots + f_nx_n)^2$.

[8] The detailed proof including the TRS is available at `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=415507`

However, for computing the values of f_i we need an executable algorithm to compute square roots of integers and rationals. To this end, we formalized variants of the Babylonian method to efficiently compute square roots. Afterwards, indeed all bounded increase proofs from AProVE could be certified by CeTA (version 2.10).

The original Babylonian algorithm is an instance of Newton's method. It approximates $\sqrt{n}$ by iteratively computing $x_{i+1} := \frac{\frac{n}{x_i}+x_i}{2}$ until x_i^2 is close enough to n. Although we did not require it—it does not deliver precise results—for completeness we formalized it, where the domain is some arbitrary linearly ordered field of type $'a$. Here, the main work was to turn the convergence proof into a termination proof. For example, we had to solve the problems that the value to which the function converges is not necessarily a part of the domain: consider $\sqrt{2}$ and $'a$ being the type of rationals.

The precise algorithms are based on the following adaptation. In order to compute square roots of integers we simply use integer divisions $x \div y = \lfloor \frac{x}{y} \rfloor$. Moreover, the result is returned as an option-type, i.e., either we return some number (the square-root), or we return nothing, indicating that there is no integer x such that $x^2 = n$.

$$\begin{aligned} \textit{int-main}\ x\ n = (&\textsf{if}\ x < 0 \vee n < 0\ \textsf{then}\ \textit{None}\ \textsf{else}\ (\textsf{if}\ x^2 \le n \\ &\textsf{then}\ (\textsf{if}\ x^2 = n\ \textsf{then}\ \textit{Some}\ x\ \textsf{else}\ \textit{None}) \\ &\textsf{else}\ \textit{int-main}\ ((n \div x + x) \div 2)\ n)) \end{aligned}$$

Termination can be proven more easily by just using x as measure. Soundness is also trivially proven as we only return *Some* x if $x^2 = n$. Indeed, the hardest part was to prove completeness which is stated as follows.

Theorem 13. $x \ge 0 \implies x^2 = n \implies y^2 \ge n \implies y \ge 0 \implies n \ge 0 \implies$ *int-main* $y\ n =$ *Some* x.

To this end, we had to show that once the value of x^2 is below n, then there is no solution. Here, in the proof the non-trivial inequality $(x^2 \div y) \cdot y + y^2 \ge 2xy$ occurred. It trivially holds if one used standard division instead of integer division. However, it took quite a while to prove the desired inequality where we first ran into several dead ends as we tried to use induction on x or y. The solution was to express x^2 as $(y-x)^2+y\cdot(2x-y)$ and then divide both summands by y, cf. the detailed proof in `Sqrt-Babylon.thy`.

Using soundness and completeness of *int-main*, it was easy to write an algorithm *sqrt-int* which invokes *int-main* with a suitable starting value of x (larger than $\sqrt{n}$) and performs a case analysis on whether n is 0, positive, or negative.

Theorem 14. *set* (*sqrt-int* n) $= \{x :: \textit{int}.\ x^2 = n\}$.

In a similar way we construct a square-root algorithm *sqrt-nat* for the naturals, where *int-main* is invoked in combination with conversion functions between naturals and integers.

Theorem 15. *set* (*sqrt-nat* n) $= \{x :: \textit{nat}.\ x^2 = n\}$.

Note, that since *int-main* only works on the positive integers, it sounds more sensible to directly implement it over the naturals. Then conditions like $x \geq 0$ can just be eliminated. Actually, when we started our formalization, we followed this approach. But it turned out that it was by far more cumbersome to perform arithmetic reasoning on the naturals than on the integers. The reason was that differences like $x - y + z + y$ easily simplify to $x + z$ over the integers, but require side-conditions like $x \geq y$ on the naturals. And the amount of required side-conditions was by far larger than storing that certain values are non-negative.

We also formalized an algorithm *sqrt-rat* where we used known facts on co-primality from the Isabelle distribution. In brief, given some rational number $\frac{p}{q}$ for integers p and q, *sqrt-rat* returns $\pm\frac{\sqrt{p}}{\sqrt{q}}$, if this is well-defined, or nothing.

Theorem 16. *set* (*sqrt-rat* n) $= \{x :: rat.\ x^2 = n\}$.

Note that using Thm. 16 one can easily figure out that $\sqrt{2}$ is an irrational number, just by evaluating that *sqrt-rat* 2 is the empty list. Moreover, since the Babylonian algorithm is efficient, it also is no problem to check that $\sqrt{12345678901234567890123456789012345678901234567890}$ is irrational.

7 Summary

We formalized the termination technique of bounded increase. To this end, in addition to the pen-and-paper proof we had to prove the missing fact that signature extensions are sound for innermost rewriting. Moreover, we not only showed that the "obvious reason" for soundness of the induction rule is wrong, but additionally provided a condition under which this obvious reason is sound: unique innermost normalization. This property follows from weak orthogonality, and our formalization contains—as far as we know—the first mechanized proof of the fact that weak orthogonality implies confluence. For the certification algorithm we also required some algorithm to precisely compute square roots, where we adapted the Babylonian approximation algorithm to obtain precise algorithms for the naturals, integers, and rationals.

All variants for computing square roots ($\approx$ 700 lines) have been made available in the archive of formal proofs, and the remaining formalization ($\approx$ 4,700 lines) is available at `http://cl-informatik.uibk.ac.at/software/ceta/`.

Acknowledgments. We thank Cezary Kaliszyk for his help on Nominal Isabelle and we thank the anonymous referees for their helpful comments and remarks.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
2. Baader, F., Nipkow, T.: Term Rewriting and All That, Cambridge (1998)
3. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 131–146. Springer, Heidelberg (2009)

4. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL - lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 19–36. Springer, Heidelberg (1999)
5. Blanqui, F., Koprowski, A.: COLOR: A Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Mathematical Structures in Computer Science 21(4), 827–859 (2011)
6. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Automated certified proofs with CiME3. In: Proc. RTA 2011. LIPIcs, vol. 10, pp. 21–30 (2011)
7. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Transactions on Programming Languages and Systems 33(2), 7:1–7:39 (2011)
8. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
9. Giesl, J., Thiemann, R., Swiderski, S., Schneider-Kamp, P.: Proving Termination by Bounded Increase. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 443–459. Springer, Heidelberg (2007) Proofs and examples available in technical report AIB-2007-03, http://aib.informatik.rwth-aachen.de
10. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
11. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
12. Hong, H., Jakuš, D.: Testing positiveness of polynomials. Journal of Automated Reasoning 21(1), 23–38 (1998)
13. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)
14. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning 44(4), 303–336 (2010)
15. Krauss, A., Sternagel, C., Thiemann, R., Fuhs, C., Giesl, J.: Termination of Isabelle functions via termination of rewriting. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 152–167. Springer, Heidelberg (2011)
16. Lankford, D.: On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA (1979)
17. Sternagel, C., Thiemann, R.: Signature extensions preserve termination. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 514–528. Springer, Heidelberg (2010)
18. Sternagel, C., Thiemann, R.: Certification of nontermination proofs. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 266–282. Springer, Heidelberg (2012)
19. Thiemann, R.: Executable Transitive Closures. In: The Archive of Formal Proofs (February 2012), http://afp.sf.net/entries/Transitive-Closure-II.shtml
20. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009)
21. Urban, C.: Nominal reasoning techniques in Isabelle/HOL. Journal of Automated Reasoning 40(4), 327–356 (2008)
22. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in Nominal Isabelle. Logical Methods in Computer Science 8(2) (2012)

Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types

Vincent Rahli, Mark Bickford, and Abhishek Anand

Cornell University, Ithaca, NY, USA

Abstract. This paper extends the proof methods used by the Nuprl proof assistant to reason about the computational behavior of its untyped programs. We have implemented new methods to prove non-trivial bisimulations between programs and have successfully applied these methods to formally optimize distributed programs such as our synthesized and verified version of Paxos, a widely used protocol to achieve software based replication. We prove new results about the basic computational equality relation on terms, and we extend the theory of partial types as the basis for stating internal results about the computation system that were previously treated only in the meta theory of Nuprl. All the lemmas presented in this paper have been formally proved in Nuprl.

1 Introduction

This paper presents proof techniques implemented in the Nuprl proof assistant [16,27,4] to reason about its own computation system and programming language, an applied lazy (call-by-name) λ-calculus. Since the computation system is universal (Turing complete), we need to reason using *partial types* introduced by Constable and Smith [35,17] and extended by Crary [18].[1] The *bisimulation* relation defined by Howe turned out to form a contextual equivalence relation [23,24], and is therefore the basic computational equality on Nuprl terms. Internally it becomes the equality on the partial type `Base` of all untyped Nuprl terms, both programs and data. The canonical values of this type are the terminating terms, the values of the type system.

Nuprl's logic is defined on top of this computation system. It is an extensional Constructive Type Theory (CTT) [16] which relies on ternary partial equivalence relations that express when two terms are equal in a type. For example, the type $1+1 =_{\mathbb{N}} 2$ expresses that $1+1$ and 2 are equal natural numbers (we write $x \in T$, for $x =_T x$). Each type is defined by such a relation.

Over the past two decades much progress has been made to enrich Nuprl and make it a practical programming language as well as a logical system in which one can verify properties of Nuprl programs [35,17,18,21,25,26]. During that period, Nuprl's theory was extended with, e.g., intersection types, union types, partial types, a call-by-value operator, rules to reason about computation, and in particular rules about the fixpoint operator. Recently, we have extended Nuprl

[1] Crary gave a denotational semantics for an ML dialect using partial types.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 261–278, 2013.

with new operators called *canonical form tests* (similar to Lisp and Scheme's type predicates) so that programs can distinguish between primitive canonical operators such as the pair or lambda constructors, and we have developed new ways to reason about these new constructs. This gives us more tools to program in Nuprl and reason about these programs.

Nuprl's intersection and partial types add expressive power. They allow us to reason about a larger class of practical programs and express more program properties. However, using typed equivalences to transform programs can be unnecessarily complex because programs are not annotated with types and both type checking and type inference are undecidable in Nuprl. Instead, we can reason about untyped program equivalences (e.g., between partial functions), which are easier to use because they only require trivial type reasoning. Such equivalences are highly useful for program transformation such as program optimization.

Using untyped reasoning, we have proved many bisimulations involving data structures such as lists. We have also used these techniques in our work on process synthesis [11,33], where processes are defined as recursive functions of a co-recursive type. Our synthesized processes were initially too slow to be used in industrial strength systems. In response to that issue, we have developed a proof methodology to simplify and optimize them. We have applied that methodology to various synthesized consensus protocols such as 2/3-consensus [14] or Multi-Paxos [28], and observed a significant speed-up. These synthesized consensus protocols have successfully been used in a replicated database [34]. This paper illustrates these proof techniques using a simple running example: appending the empty list to a term. It then illustrates their use to optimize distributed processes synthesized from protocol specifications.

Finally, being able to reason about Nuprl's programming language directly in Nuprl is another step towards a longstanding goal of building a correct-by-construction, workable logical programming environment [22]. An obvious question is then, could we build a verified compiler for Nuprl in Nuprl that generates reasonably fast code? Modern proof assistants that implement constructive type theories such as Coq [9,1], Isabelle [8,7], or Nuprl rely on unverified compilers. Even though the programs they generate, e.g., by extraction from proofs, are *correct-by-construction*, one could argue whether the machine code obtained after compilation is still correct. Thus, we would like these proof assistants to be expressive enough to program and verify optimized compilers for their underlying programming languages, and to program these proof assistants in themselves.

The contributions of this paper are as follows: (1) we introduce new formal untyped reasoning techniques for proving bisimulations, which expose more of the computation system to formal reasoning; and (2) we apply these techniques to optimize distributed processes.

$v ::= \underline{n}$	(integer)	$\mid \lambda x.t$	(lambda)	
$\mid \langle t_1, t_2 \rangle$	(pair)	$\mid \texttt{inl}(t)$	(left injection)	
$\mid \texttt{inr}(t)$	(right injection)	$\mid \texttt{Ax}$	(axiom)	
$t ::= x$	(variable)	$\mid \texttt{isaxiom}(\boxed{t_1}, t_2, t_3)$	(isaxiom)	
$\mid v$	(value)	$\mid \texttt{ispair}(\boxed{t_1}, t_2, t_3)$	(ispair)	
$\mid \boxed{t_1}\ t_2$	(application)	$\mid \texttt{islambda}(\boxed{t_1}, t_2, t_3)$	(islambda)	
$\mid \texttt{fix}(\boxed{t})$	(fixpoint)	$\mid \texttt{isinl}(\boxed{t_1}, t_2, t_3)$	(isinl)	
$\mid \texttt{let}\ x := \boxed{t_1}\ \texttt{in}\ t_2$	(call-by-value)	$\mid \texttt{isinr}(\boxed{t_1}, t_2, t_3)$	(isinr)	
$\mid \texttt{let}\ x ::= \boxed{t_1}\ \texttt{in}\ t_2$	(call-by-valueall)	$\mid \texttt{isint}(\boxed{t_1}, t_2, t_3)$	(isint)	
$\mid \texttt{let}\ x, y = \boxed{t_1}\ \texttt{in}\ t_2$	(spread)			
$\mid \texttt{case}\ \boxed{t_1}\ \texttt{of inl}(x) \Rightarrow t_2 \mid \texttt{inr}(y) \Rightarrow t_3$	(decide)			
$\mid \texttt{if}\ \boxed{t_1} = \boxed{t_2}\ \texttt{then}\ t_3\ \texttt{else}\ t_4$	(integer equality)			

Fig. 1. Syntax of Nuprl's programming language

2 Nuprl's Programming Language

2.1 Syntax

Nuprl is defined on top of an applied lazy untyped λ-calculus. Fig. 1 introduces a subset of this language, where $\underline{n}$ ranges over integers. Because this language is lazy, its values[2] (or canonical forms) are either integers, lambda abstractions, pairs, injections, or `Ax`. The canonical form `Ax` (sometimes written as $\star$) is the unique canonical inhabitant of true propositions that do not have any nontrivial computational meaning in CTT, such as $0 =_{\mathbb{N}} 0$ which is an axiom of the logic. Non-canonical terms (non-values) have arguments that are said to be *principal*. These principal arguments indicate which subterms of a non-canonical term have to be evaluated before checking whether the term itself is a redex or not. Principal arguments of terms are marked with boxes in the above table. In the rest of this paper, variables will be obvious from the context (we often use x and y such as in Fig. 1), we use v for values, and the other letters can be any term. When it is more readable we write $t_1(t_2)$ instead of $t_1\ t_2$.

As mentioned above, we have recently added new primitive operators to Nuprl: the canonical form tests such as `ispair`. Adding these primitive forms was a design decision we made to distinguish between canonical forms (e.g., see `list_ind`'s definition below) and therefore exploit Howe's bisimulation even further. Our experiments with them have proven to be very fruitful.

Let us now define a few useful abstractions: let $\bot$ (bottom) be $\texttt{fix}(\lambda x.x)$, let $\pi_1(t)$ be $(\texttt{let}\ x, y = t\ \texttt{in}\ x)$, and let $\pi_2(t)$ be $(\texttt{let}\ x, y = t\ \texttt{in}\ y)$.

Free and bound variables are defined as usual. We write $t[x \backslash u]$ (and more generally $t[x_1 \backslash u_1; \cdots; x_n \backslash u_n]$) for the term t in which all the free occurrences of x have been replaced by u. Terms are identified up to alpha-equivalence.

[2] The only other values currently in Nuprl are tokens, atoms, and types, but more values can be added because the system is open-ended.

Core calculus:

$(\lambda x.F)\ a$	$\rightarrow$	$F[x\backslash a]$
$\texttt{let}\ x, y = \langle t_1, t_2\rangle\ \texttt{in}\ F$	$\rightarrow$	$F[x\backslash t_1; y\backslash t_2]$
$\texttt{case inl}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G$	$\rightarrow$	$F[x\backslash t]$
$\texttt{case inr}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G$	$\rightarrow$	$G[y\backslash t]$
$\texttt{if}\ \underline{n}_1 = \underline{n}_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	$\rightarrow$	t_1, if $\underline{n}_1 = \underline{n}_2$
$\texttt{if}\ \underline{n}_1 = \underline{n}_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	$\rightarrow$	t_2, if $\underline{n}_1 \neq \underline{n}_2$
$\texttt{fix}(t)$	$\rightarrow$	$t\ \texttt{fix}(t)$
$\texttt{let}\ x := t_1\ \texttt{in}\ t_2$	$\rightarrow$	$t_2[x\backslash t_1]$, if t_1 is a value

Canonical form tests:

$\texttt{ispair}(\langle t, t'\rangle, t_1, t_2)$	$\rightarrow t_1$	$\texttt{ispair}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a pair
$\texttt{isaxiom}(\texttt{Ax}, t_1, t_2)$	$\rightarrow t_1$	$\texttt{isaxiom}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not axiom
$\texttt{islambda}(\lambda x.t, t_1, t_2)$	$\rightarrow t_1$	$\texttt{islambda}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a lambda
$\texttt{isinl}(\texttt{inl}(t), t_1, t_2)$	$\rightarrow t_1$	$\texttt{isinl}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a left injection
$\texttt{isinr}(\texttt{inr}(t), t_1, t_2)$	$\rightarrow t_1$	$\texttt{isinr}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not a right injection
$\texttt{isint}(\underline{n}, t_1, t_2)$	$\rightarrow t_1$	$\texttt{isint}(v, t_1, t_2)$	$\rightarrow t_2$, if v is not an integer

Fig. 2. Nuprl's operational semantics

Let `Top` be the following type: for all closed terms t_1 and t_2, $t_1 =_{\texttt{Top}} t_2$. `Top`'s equality is trivial because it identifies all elements. This type is especially useful to assign types to terms in contexts where their structure or behavior is irrelevant. When discussing types it is important to remember that a type is an equivalence relation on a set of terms and not simply a set of terms. Type A is a subtype of type B (written $A \sqsubseteq B$) if $x =_A y$ implies $x =_B y$. This means not only that every term in A is also in B, but that equality in A refines equality in B. Hence, $T \sqsubseteq \texttt{Top}$ for every type T. Sec. 3.1 discusses the type `Base`, which contains all Nuprl terms, but does not have this property (i.e. not every type T is a subtype of `Base`[3]) because equality on `Base` is Howe's bisimulation relation.

2.2 Operational Semantics

Fig. 2 presents some of Nuprl's reduction rules. This figure does not show the reduction rule for the call-by-valueall operator because it is slightly more complicated. This operator is like call-by-value but continues recursively evaluating subterms of pairs and injections.[4]

At any point in a computation, either a value is produced, or the computation is stuck, or we can take another step. For example, $(\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F)$ is a meaningless term that cannot evaluate further. It is stuck on the wrong kind of

[3] Being extentional, function types are in general not subtypes of `Base`.

[4] The call-by-valueall operator is similar to a restricted form of Haskell's *deepseq* operator. It can be defined using the other primitive operators (see the expanded version of this article at `http://www.nuprl.org/Publications/`), but for simplicity reasons we introduce it as a primitive in this paper.

principal argument: Ax instead of a pair. Using the proof techniques presented below, in Sec. 4.3 we prove that this term is computationally equivalent to $\bot$. We can prove such results using ispair and isaxiom, and do not know of any other way discussed in the literature to accomplish this. Intuitively, we prove this lemma using the fact that isaxiom can compute to different values depending on whether its first argument computes to Ax or not. For example, $\texttt{isaxiom}(t, 0, 1)$ reduces to 0 if t is Ax and to 1 if t is, e.g., a pair. Note that even though they are computationally equal, ($\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F$) and $\bot$ are fundamentally different in the sense that one could potentially detect whether a term is stuck (by slightly modifying our destructors such as *spread* or *decide*), but one cannot detect whether a term diverges or not.

2.3 Datatypes

Booleans. As usual, we define Booleans using the disjoint union type as follows: $\mathbb{B} = \texttt{Unit} + \texttt{Unit}$. The Unit type is defined as $0 =_{\mathbb{Z}} 0$ and therefore Ax is its only inhabitant (up to computation). We define the Boolean true tt as $\texttt{inl}(\texttt{Ax})$, and the Boolean false ff as $\texttt{inr}(\texttt{Ax})$. Using the *decide* operator we define a conditional operator as follows: $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 = \texttt{case}\ t_1\ \texttt{of}\ \texttt{inl}(x) \Rightarrow t_2 \mid \texttt{inr}(x) \Rightarrow t_3$.

Lists. We define lists as follows using Nuprl's union type [26] and recursive type [32] that allows one to build inductive types:[5] $\texttt{List}(T) = \texttt{rec}(L.\texttt{Unit} \cup T \times L)$. The type constructor $\cup$ creates the union of two types, not the disjoint union. The members of $A \cup B$ are members of A or B, not injections of them. A list is either a member of Unit, i.e., Ax, or a pair. The empty list nil is defined as Ax, and the cons operation, denoted by $\bullet$, as the pair constructor. We can distinguish an empty list and a non empty list because Unit and the product type are disjoint. Using fix, we define the following "list induction" operator:

$$\texttt{list\_ind}(L, b, f) =$$
$$\texttt{fix}(\lambda F.\lambda L.\texttt{ispair}(L, \texttt{let}\ h, t = L\ \texttt{in}\ f\ h\ t\ (F\ t), \texttt{isaxiom}(L, b, \bot)))\ L$$

To define such a function that takes a list as input, we need to be able to test whether it is a pair or Ax. If we were to use the *spread* operator, we could destruct pairs, but computations would get stuck on Ax which we use to represent the empty list. Therefore, we need an operator such as the ispair canonical form test which allows us to perform two different computations depending on whether its first argument computes to a pair or not. Note that if list_ind's first argument does not compute to a pair or to Ax, then the term diverges as opposed to returning an arbitrary value. This is necessary to prove untyped equivalences between list operations. We define the append and map operations as follows:

$$t_1\ \texttt{@}\ t_2 \quad = \texttt{list\_ind}(t_1, t_2, \lambda h.\lambda t.\lambda r.h \bullet r)$$
$$\texttt{map}(f, t) = \texttt{list\_ind}(t, \texttt{nil}, \lambda h.\lambda t.\lambda r.(f\ h) \bullet r)$$

[5] This new definition of lists replaces the one from Nuprl's book [16] where lists are considered as primitive objects. Using Nuprl's replay functionality, we were able to successfully replay the entire Nuprl library using this new definition of lists.

3 Computational Equivalence

3.1 Simulations and Bisimulations

Howe [23,24] defined the simulation or approximation relation $\leq$ using the following co-inductive rule: $t_1 \leq t_2$ if and only if (if t_1 computes to a canonical form $\Theta(u_1, \ldots, u_n)$ of the language defined in Sec. 2.1, then t_2 computes to a canonical form $\Theta(u'_1, \ldots, u'_n)$ such that for all $i \in \{1, \ldots, n\}$, $u_i \leq u'_i$). We say that t_1 approximates t_2 or that t_2 simulates t_1. This relation is reflexive (w.r.t. the terms defined in Sec. 2.1) and transitive. Howe then defined the bisimulation relation $\sim$ as the symmetric closure of $\leq$ (i.e., $t_1 \sim t_2$ iff $t_1 \leq t_2$ and $t_2 \leq t_1$), and proved that $\leq$ and $\sim$ are congruences w.r.t. Nuprl's computation system.[6]

The following *context property* follows from the fact that $\sim$ is a congruence:

$$\forall i : \{1, \ldots, n\}.\ t_i \leq u_i \ \Rightarrow\ G[x_1 \backslash t_1; \cdots; x_n \backslash t_n] \leq G[x_1 \backslash u_1; \cdots; x_n \backslash u_n]$$

Howe's bisimulation relation respects computation, i.e., if $t_1 \sim t_2$ then (1) t_1 computes to a value iff t_2 computes to a value, and (2) if t_1 computes to a value v_1 then t_2 computes to a value v_2 with same outer operator such that $v_1 \sim v_2$.

Because $\bot$ does not compute to a canonical form, by definition $\bot \leq t$ is true for any term t; hence for example $\langle \mathtt{Ax}, \bot \rangle \leq \langle \mathtt{Ax}, \mathtt{Ax} \rangle$. Similarly, because $\mathtt{Ax}$ is not a pair, ($\mathtt{let}\ x, y = \mathtt{Ax}\ \mathtt{in}\ x$) does not compute to a canonical form, and by definition, $\mathtt{let}\ x, y = \mathtt{Ax}\ \mathtt{in}\ x \leq t$ is true for any term t (we prove $\mathtt{let}\ x, y = \mathtt{Ax}\ \mathtt{in}\ F \sim \bot$ in Sec. 4.3). However, $\mathtt{Ax} \leq \bot$ is not true because $\bot$ diverges while $\mathtt{Ax}$ is a value; hence $\langle \mathtt{Ax}, \mathtt{Ax} \rangle \leq \langle \mathtt{Ax}, \bot \rangle$ is not true either.

Let us write $\mathtt{halts}(t)$ if t reduces to a value—we say that t converges. We can define convergence using call-by-value because the call-by-value operator ($\mathtt{let}\ x := t_1\ \mathtt{in}\ t_2$) first evaluates t_1. The term t_1 converges if and only if the term ($\mathtt{let}\ x := t_1\ \mathtt{in}\ \mathtt{Ax}$) evaluates to $\mathtt{Ax}$. So we simply define $\mathtt{halts}(t)$ to be the simulation $\mathtt{Ax} \leq (\mathtt{let}\ x := t\ \mathtt{in}\ \mathtt{Ax})$. Because $\mathtt{Ax}$ is a canonical value then $\mathtt{Ax} \leq (\mathtt{let}\ x := t\ \mathtt{in}\ \mathtt{Ax})$ is true if and only if ($\mathtt{let}\ x := t\ \mathtt{in}\ \mathtt{Ax}$) computes to $\mathtt{Ax}$, i.e., if and only if t computes to a value.

Constable and Smith [35,17] introduced partial types to reason about computations that might not halt. For any type T, the partial type $\overline{T}$ contains all members of T as well as all divergent terms, and has the following equality: two terms are equal in $\overline{T}$ if they have the same convergence behavior (i.e., either neither computes to a value or both compute to a value), and when they converge, they are equal in T. An important partial type is $\mathtt{Base} = \overline{\mathtt{Value}}$ where $\mathtt{Value}$ is the type of all closed canonical terms of the computation system with $\sim$ as its equality. Because $\mathtt{Base}$ is a partial type, it contains converging as well as diverging terms, and equal terms have the same convergence behavior.

[6] Howe proved that $\sim$ is a congruence w.r.t. a lazy computation system by proving that all the operators of the system satisfy a property called *extensionality*. The expanded version of this article proves that the new operators introduced in this paper satisfy that property.

3.2 Simple Facts about Lists

Sec. 4 proves that for all terms f and t in Top, $\texttt{map}(f, t)$ @ nil $\sim \texttt{map}(f, t)$.

If t is a list, the first expression ($\texttt{map}(f, t)$ @ nil) requires two passes over the list t while the second expression ($\texttt{map}(f, t)$) requires only one. This simple bisimulation will be our running example to illustrate the techniques we use to optimize our distributed processes (discussed in Sec. 5).

Note that this lemma would be easy to prove by induction on the list t if we were using the list type instead of Top. However, we might need to instantiate t with a term for which it would be non-trivial to prove that it is a list because Nuprl is based on an extension of the *untyped* λ-calculus and type inference and type checking are *undecidable*. In addition, if we were to use a typed equality (instead of $\sim$) for substitution in some context, then we would also have to prove that the context is functional over the type of the equality. That is, to rewrite in the term $C[t]$ of type B using $t =_A u$, we have to prove that $\lambda z.C[z]$ is of type $A \to B$. Moreover, the above equivalence is indeed true for any term t, e.g., it is true when t is a stream.

Note that it is not true that for all terms t, t @ nil $\sim t$. For example, by definition of @, $(\lambda x.x)$ @ nil $\sim \bot$. However, the bisimulation $\lambda x.x \sim \bot$ is not true because the simulation $\lambda x.x \leq \bot$ is not true. This shows that there are some terms t for which $t \leq t$ @ nil does not hold.[7] However, Sec. 4 proves that for all terms t, t @ nil $\leq t$. A corollary of that lemma is that $\texttt{map}(f, t)$ @ nil $\leq \texttt{map}(f, t)$.

4 Proof Techniques

This section presents three proof techniques we use to prove bisimulations: Crary's least upper bound property [18], patterns of reasoning regarding our new canonical form tests, and patterns of reasoning regarding our halts operator. It also presents three derived proof techniques called lifting, normalization, and strictness. Using these techniques, we prove $\texttt{map}(f, t)$ @ nil $\sim \texttt{map}(f, t)$, and in Sec. 5, we optimize distributed processes.

4.1 Least Upper Bound Property

Using the properties of $\leq$ and that $\texttt{fix}(f) = f\ \texttt{fix}(f)$, it is easy to prove by induction on n that $\forall n : \mathbb{N}.\ f^n(\bot) \leq \texttt{fix}(f)$ [18]. So $\texttt{fix}(f)$ is an upper bound of its approximations. The least upper bound property [18, Theorem 5.9] is:

Rule [least-upper-bound]. $\forall n : \mathbb{N}.\ G(f^n(\bot)) \leq t \ \Rightarrow\ G(\texttt{fix}(f)) \leq t$.

[7] The expanded version of this article provides a characterization of the terms that satisfy that property.

4.2 Canonical Form Tests

In order to reason about its programs, we gave NuPRL the ability to reason about the *canonical form tests* such as `ispair`, `isaxiom`, etc.[8] These effective operations on `Base` allow us to reason in the programming language, where in the past we resorted to reflection in the logic [6].

Membership Rules

Rule `[ispair-member]`. *To prove that* $\texttt{ispair}(t_1, t_2, t_3) \in T$, *it is enough to prove* $\texttt{halts}(t_1)$, *and that both* t_2 *and* t_3 *are members of* T.

We introduce similar rules for the other canonical form tests. Using this rule we can trivially prove the following fact:

Lemma 1. *For all terms* t *in* `Base`, *if* $\texttt{halts}(t)$ *then* $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \in \mathbb{B}$.

The same is true for the other tests. Using these facts, we can, e.g., decide whether a converging term is a pair or not.

Semi-decision Rules. Depending on how `ispair` computes we can deduce various pieces of information. If we know that $\texttt{ispair}(t_1, t_2, t_3)$ always computes to t_2 and cannot compute to t_3 then we know that t_1 is a pair. If we know that $\texttt{ispair}(t_1, t_2, t_3)$ always computes to t_3 and cannot compute to t_2 then we know that t_1 is not a pair. These properties are captured by the following two rules:

Rule `[ispair]`. *To prove* $t \in \texttt{Top} \times \texttt{Top}$ *(i.e., t is a pair), it is enough to prove* $\texttt{ispair}(t, \texttt{inl}(a), \texttt{inr}(b)) \sim \texttt{inl}(a)$ *for some terms a and b.*

Rule `[not-ispair]`. *To prove* $\texttt{ispair}(t_1, t_2, t_3) \sim t_3$, *it is enough to prove that* $\texttt{ispair}(t_1, \texttt{inl}(a), \texttt{inr}(b)) \sim \texttt{inr}(b)$ *for some terms a and b.*

We introduce similar rules for the other canonical form tests. Using these rules we can prove such results as (similar results are true for the other tests):

Lemma 2. *For all terms t, a, b in* `Base`, *if* $\texttt{halts}(t)$ *then* $t \sim \langle \pi_1(t), \pi_2(t) \rangle \vee \texttt{ispair}(t, a, b) \sim b$.

Proof. By Lemma 1, $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \in \mathbb{B}$. Therefore, either $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{tt}$ or $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{ff}$ (this is true for any Boolean). If $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{tt}$ then using rule `[ispair]` we obtain that t is a pair and therefore $t \sim \langle \pi_1(t), \pi_2(t) \rangle$. If $\texttt{ispair}(t, \texttt{tt}, \texttt{ff}) \sim \texttt{ff}$ then using rule `[not-ispair]` we obtain that $\texttt{ispair}(t, a, b) \sim b$. □

[8] The proofs that the rules introduced in this section are valid w.r.t. Allen's PER (Partial Equivalence Relations) semantics [2,3] are presented in the expanded version of this article.

```
⊢ ∀F:Top. (let x,y = Ax in F[x;y] ~ bottom())
|
BY (SqReasoning
    THEN Assert ⌜(if Ax is a pair then 0 otherwise 1) = 1⌝⋅
    THEN (Reduce 0 THEN Auto THEN AutoPairEta [2;1] (-1)))
```

Fig. 3. Computational equivalence between $\bot$ and a stuck term

4.3 Convergence

Rule `[convergence]`. *To prove* $t_1 \leq t_2$*, one can assume* $\texttt{halts}(t_1)$.

This rule follows directly from $\leq$'s definition. For example, to prove $\texttt{let}\ x, y = p\ \texttt{in}\ F \leq \texttt{let}\ x, y = q\ \texttt{in}\ G$, one can assume that $\texttt{halts}(\texttt{let}\ x, y = p\ \texttt{in}\ F)$.

Nuprl also has rules to reason about $\texttt{halts}(t)$. If a non-canonical term converges, then its principal arguments have to converge to the appropriate canonical forms as presented in Fig 2. For example the following two rules follow from the operational semantics of *spread* and `ispair` (we have similar rules for the other non-canonical operators):

Rule `[halt-spread]`. *If* $\texttt{halts}(\texttt{let}\ x, y = p\ \texttt{in}\ F)$ *then* p *computes to a pair.*

Rule `[halt-ispair]`. *If* $\texttt{halts}(\texttt{ispair}(t_1, t_2, t_3))$ *then* $\texttt{halts}(t_1)$.

Let us go back to the example presented in Sec. 2.2. We now have enough tools to prove the following lemma:

Lemma 3. *For all terms* F *in* `Top`, $\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F \sim \bot$

Proof. Fig 3 presents our Nuprl proof of that fact. That proof goes as follows: By definition of $\sim$, we have to prove $\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F \leq \bot$ and $\bot \leq \texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F$. The second simulation is trivial. Let us prove the first one. Using `[convergence]`, we can assume $\texttt{halts}(\texttt{let}\ x, y = \texttt{Ax}\ \texttt{in}\ F)$ and using `[halt-spread]`, that `Ax` is a pair. This reasoning is done by our `SqReasoning` tactic. Finally, the term $\texttt{ispair}(\texttt{Ax}, 0, 1)$ computes to 1, and because we deduced that `Ax` is a pair, it also reduces to 0, and we have an absurdity. □

4.4 Lifting

Now we describe the following derived proof techniques: lifting, normalization (see Sec. 4.5 below), and strictness (see Sec. 4.6 below) which are used in Sec. 4.7 below. Lifting transforms a term t into t' such that $t \sim t'$ and such that t' has a smaller path to the principal argument of a subterm of t. Let us now provide a few examples. The following bisimulation specifies a lifting operation, where the path to p is shorter in the second term than in the first term:

Lemma 4. *For all terms* F *and* G *in* `Top`*:*

$$\texttt{let}\ c, d = (\texttt{let}\ a, b = p\ \texttt{in}\ F)\ \texttt{in}\ G \sim \texttt{let}\ a, b = p\ \texttt{in}\ (\texttt{let}\ c, d = F\ \texttt{in}\ G)$$

Proof. To prove that bisimulation, we have to prove that the first term simulates the second one and vice versa. Let us prove that the second one simulates the first one (the other direction is similar), i.e., $\texttt{let}\ c, d = (\texttt{let}\ a, b = p\ \texttt{in}\ F)\ \texttt{in}\ G \leq \texttt{let}\ a, b = p\ \texttt{in}\ (\texttt{let}\ c, d = F\ \texttt{in}\ G)$. Using `[convergence]`, we can assume $\texttt{halts}(\texttt{let}\ c, d = (\texttt{let}\ a, b = p\ \texttt{in}\ F)\ \texttt{in}\ G)$, from which, using `[halt-spread]` twice, we obtain that p is a pair. More precisely, we can prove that p is the pair $\langle \pi_1(p), \pi_2(p) \rangle$. By replacing p by $\langle \pi_1(p), \pi_2(p) \rangle$ in the above simulation, and by reducing both sides, we obtain $\texttt{let}\ c, d = F[a \backslash \pi_1(p); b \backslash \pi_2(p)]\ \texttt{in}\ G \leq \texttt{let}\ c, d = F[a \backslash \pi_1(p); b \backslash \pi_2(p)]\ \texttt{in}\ G$, which is true by reflexivity of $\leq$. □

Using this lemma, one can, e.g., derive the following chain of rewrites:

$$\begin{array}{l} \texttt{let}\ a, b = (\texttt{let}\ c, d = p\ \texttt{in}\ \langle c, d \rangle)\ \texttt{in}\ F \\ \sim \texttt{let}\ c, d = p\ \texttt{in}\ (\texttt{let}\ a, b = \langle c, d \rangle\ \texttt{in}\ F) \\ \quad \sim \texttt{let}\ c, d = p\ \texttt{in}\ F[a \backslash c; b \backslash d] \end{array}$$

The following bisimulation specifies another lifting operation where the path to t_1 is shorter in the second term than in the first one:

Lemma 5. *For all terms t_1, t_2, t_3, t_4, and t_5 in* `Top`*:*

$$\begin{array}{l} \texttt{ispair}(\texttt{ispair}(t_1, t_2, t_3), t_4, t_5) \\ \sim \texttt{ispair}(t_1, \texttt{ispair}(t_2, t_4, t_5), \texttt{ispair}(t_3, t_4, t_5)) \end{array}$$

The proof of this is similar to the proof of Lemma 4. Because lifting does not always result in a smaller term it must therefore be used in a controlled way.

4.5 Normalization

Normalization allows one to make use of the information given by destructors such as spread or decide, i.e., that some terms are forced to be pairs or injections by the computation system. Normalization achieves some kind of common subexpression elimination, which is a standard optimization technique. For example, the next lemma says that the expression on left-hand-side has a value if and only if p (which can be an arbitrary large term) is a pair, and more precisely in F it has to be the pair $\langle a, b \rangle$:

Lemma 6. *For all terms p and F in* `Top`*:*

$$\texttt{let}\ x, y = p\ \texttt{in}\ F[z \backslash p] \sim \texttt{let}\ x, y = p\ \texttt{in}\ F[z \backslash \langle x, y \rangle]$$

The proof of this is similar to the proof of Lemma 4.

4.6 Strictness

Strictness says that if $\bot$ is one of the principal arguments of a term then this term is computationally equal to $\bot$. For example we proved the following lemma:

Lemma 7. *For all terms F in* Top, $(\texttt{let}\ x, y = \bot\ \texttt{in}\ F) \sim \bot$.

The proof of this is similar to the proof of Lemma 4. Intuitively, such lemmas are true because to evaluate a non-canonical term, one has to evaluate its principal arguments. If one of these principal arguments is $\bot$, then the computation diverges. Therefore, the entire term is computationally equal to $\bot$.

4.7 Back to Our List Example

As explained in Sec. 3.2, to prove $\texttt{map}(f, t)\ \texttt{@}\ \texttt{nil} \sim \texttt{map}(f, t)$, we first prove the following lemma:

Lemma 8. *For all terms t in* Top, $t\ \texttt{@}\ \texttt{nil} \leq t$.

Proof. Because `@` is defined using `fix` (see Sec. 2.3), we prove that lemma using the `[least-upper-bound]` rule (see Sec.4.1). We now have to prove that any approximation of the fixpoint is simulated by t. Let

$$F = \lambda F.\lambda L.\texttt{ispair}(L, \texttt{let}\ x, y = L\ \texttt{in}\ x \bullet (F\ y), \texttt{isaxiom}(L, \texttt{nil}, \bot))$$

We have $(t\ \texttt{@}\ \texttt{nil}) = (\texttt{fix}(F)\ t)$ by definition of append and beta-reduction. We have to prove that for all natural numbers n, and for all terms t,

$$F^n\ \bot\ t \leq t$$

which we prove by induction on n. The base case boils down to proving that $\bot\ t \leq t$ which is true using strictness. In the interesting induction case, assuming that for all terms t, $F^{n-1}\ \bot\ t \leq t$, we have to prove $F\ (F^{n-1}\ \bot)\ t \leq t$, i.e.,

$$\texttt{ispair}(t, \texttt{let}\ x, y = t\ \texttt{in}\ x \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t, \texttt{nil}, \bot)) \leq t \quad (1)$$

Let P be $\texttt{ispair}(t, \texttt{let}\ x, y = t\ \texttt{in}\ x \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t, \texttt{nil}, \bot))$. Using `[convergence]`, we can assume $\texttt{halts}(P)$. Using `[halt-ispair]`, we obtain $\texttt{halts}(t)$. By Lemma 2, we get $t \sim \langle \pi_1(t), \pi_2(t) \rangle$ or $P \sim \texttt{isaxiom}(t, \texttt{nil}, \bot)$.

If $t \sim \langle \pi_1(t), \pi_2(t) \rangle$, we have to prove the following simulation obtained from simulation 1 by replacing t by $\langle \pi_1(t), \pi_2(t) \rangle$ and by reducing:

$$\pi_1(t) \bullet ((F^{n-1}\ \bot)\ \pi_2(t)) \leq \langle \pi_1(t), \pi_2(t) \rangle$$

Because the cons operator is defined as the pair constructor, by the context property it remains to prove the following simulation, which is true by induction hypothesis: $((F^{n-1}\ \bot)\ \pi_2(t)) \leq \pi_2(t)$.

If $P \sim \texttt{isaxiom}(t, \texttt{nil}, \bot)$, we have to prove $\texttt{isaxiom}(t, \texttt{nil}, \bot) \leq t$. Using the version of Lemma 2 for `isaxiom`, we obtain $t \sim \texttt{Ax}$ or $\texttt{isaxiom}(t, \texttt{nil}, \bot) \sim \bot$. Both cases are trivial: in the first case we have to prove $\texttt{Ax} \leq \texttt{Ax}$ and in the second we have to prove $\bot \leq t$. □

Let us now prove the lemma we set out to prove in Sec. 3.2:

Lemma 9. *For all terms t and f in* Top, $\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \sim \texttt{map}(f,t)$.

Proof. By definition of $\sim$, we have to prove $\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \leq \texttt{map}(f,t)$ (which is true by Lemma 8), and $\texttt{map}(f,t) \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$. Because map is a fixpoint, we can prove the latter using the [least-upper-bound] rule. Let

$$F = \lambda F.\lambda L.\texttt{ispair}(L, \texttt{let}\ x,y = L\ \texttt{in}\ (f\ x) \bullet (F\ y), \texttt{isaxiom}(L,\texttt{nil},\bot))$$

We then have to prove that for all natural numbers n and for all terms f and t,

$$F^n\ \bot\ t \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$$

which we prove by induction on n. Once again, the base case is trivial. Assume that for all terms t, $F^{n-1}\ \bot\ t \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$, we have to prove that $F\ (F^{n-1}\ \bot)\ t \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}$, i.e., we have to prove the following simulation:

$$\begin{array}{l}\texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t,\texttt{nil},\bot)) \\ \leq \texttt{map}(f,t)\ \texttt{@}\ \texttt{nil}\end{array} \quad (2)$$

Let $P = \texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet \texttt{map}(f,y), \texttt{isaxiom}(t,\texttt{nil},\bot))$, which is $\texttt{map}(f,t)$ unfolded once. We obtain the following sequence of bisimulations by unfolding the definitions of map and @ in $(\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil})$:

$$\begin{array}{l}\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \sim P\ \texttt{@}\ \texttt{nil} \\ \sim \texttt{ispair}(P, \texttt{let}\ x,y = t\ \texttt{in}\ x \bullet (y\ \texttt{@}\ \texttt{nil}), \texttt{isaxiom}(P,\texttt{nil},\bot))\end{array}$$

Using lifting (Lemma 5) and normalization, we obtain the following bisimulation:

$$\begin{array}{l}\texttt{map}(f,t)\ \texttt{@}\ \texttt{nil} \\ \sim \texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet (\texttt{map}(f,y)\ \texttt{@}\ \texttt{nil}), \texttt{isaxiom}(t,\texttt{nil},\bot))\end{array}$$

Therefore, given that we have to prove simulation 2, it means that we have to prove the following simulation:

$$\begin{array}{l}\texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet ((F^{n-1}\ \bot)\ y), \texttt{isaxiom}(t,\texttt{nil},\bot)) \\ \leq \texttt{ispair}(t, \texttt{let}\ x,y = t\ \texttt{in}\ (f\ x) \bullet (\texttt{map}(f,y)\ \texttt{@}\ \texttt{nil}), \texttt{isaxiom}(t,\texttt{nil},\bot))\end{array}$$

which is true by induction hypothesis and the context property. □

5 Process Optimization

Nuprl implements a Logic of Events (LoE) [10,12,13] to specify and reason about distributed programs, as well as a General Process Model (GPM) [11] to implement them. We have proved a direct relationship between some LoE combinators and some GPM combinators. This allows us to automatically generate processes that are guaranteed to satisfy the logical specifications of LoE.

Using the proof techniques presented in the above section, we were able to optimize many automatically generated GPM processes. For example, we optimized our synthesized version of Paxos, which is used by the ShadowDB replicated database [34]. Because our synthesized Paxos was initially too slow, it was

only used to handle database failures, which are critical to handle correctly but are not frequent. When a failure occurs, Paxos ensures that the replicas agree on the next set of replicas. We can now also use Paxos to consistently order the transactions of the replicated databases. Initially, our synthesized code could only handle one transaction every few seconds. Thanks to our automatic optimizer, the code we synthesize is now about an order of magnitude faster. Our goal is to be able to handle several thousands of transactions per second. Even though we have not yet reached that goal, this work is already an encouraging first step towards generating fast correct-by-construction code.

A GPM process is modeled as a function that takes inputs and computes a new process as well as outputs. For distributed programs based on message passing, these inputs and outputs are messages. Formally, a process that takes inputs of type A, and outputs elements of type B, is an element of (a variant of) the following co-recursive type:

$$\texttt{corec}(\lambda P.A \to P \times \texttt{Bag}(B))$$

where `corec` is defined as follows:

$$\texttt{corec}(G) = \cap n : \mathbb{N}.\texttt{fix}(\lambda P.\lambda n.\texttt{if } n = 0 \texttt{ then Top else } G\ (P\ (n-1)))\ n$$

Note the use of bags, also called multisets, formally defined as quotiented lists. The reason for using that type is outside the scope of this paper. However, let us mention that processes can output more than one element and these elements need not be ordered. In the rest of this paper, we use curly braces to denote specific bag instances. Lists and bags have many similar operations such as: `bmap` the map operation on bags, `bnull` the null operation, `bconcat` the concat operation which flattens bags of bags, and `>>=` the bind operation of the bag monad, defined as b `>>=` f = $\texttt{bconcat}(\texttt{bmap}(f, b))$. For example, $(\{1;2;2;4\}$ `>>=` $\lambda x.\{x; x+1\}) = \{1;2;2;3;2;3;4;5\} = \{1;2;2;2;3;3;4;5\}$.

Many of the GPM combinators are defined using `fix`. Because processes are typically defined using several combinators, fixpoints end up being deeply nested which affects the computational complexity of the processes. Using, among other things, the least upper bound property, we can often reduce the number of fixpoints occurring in processes. This is our main process optimization technique.

Let us now present some GPM combinators. Processes often need to maintain an internal state. Therefore, the combinators defined below will all be of the form $\texttt{fix}(\lambda F.\lambda s.\lambda m.G)\ init$, where *init* is an initial state, and G is a transition function that takes the current state of the process (s) and an input (m), and generates a new process and some output.

5.1 Combinators

Base Combinator. It builds a process that applies a function to its inputs:

$$\texttt{base}(f) = \texttt{fix}(\lambda F.\lambda s.\lambda m.\langle F\ s, f\ m\rangle)\ \texttt{Ax}$$

Base processes are stateless, which is modeled using the term `Ax` as the state of the base combinator.

Composition Combinator. It builds a process that applies a function f to the outputs of its sub-component X:

$$f \texttt{ o } X = \texttt{fix}(\lambda F.\lambda X.\lambda m.\left(\begin{array}{l}\texttt{let } X', out = X\ m \texttt{ in}\\ \texttt{let } out' ::= \texttt{bmap}(f, out) \texttt{ in}\\ \langle F\ X', out'\rangle\end{array}\right))\ X$$

The state maintained by f `o` X is the state maintained by X. Note that for efficiency issues, we use the call-by-valueall operator ::= in order to generate the outputs out'.

Buffer Combinator. From an initial buffer *init* and a process X producing transition functions, this combinator builds a process that buffers its outputs:

$$\texttt{buffer}(X, init) =$$
$$\texttt{fix}(\lambda F.\lambda s.\lambda m.\left(\begin{array}{l}\texttt{let } X, buf = s \texttt{ in}\\ \texttt{let } X', b = X\ m \texttt{ in}\\ \texttt{let } b' ::= b \texttt{ >>= } \lambda f.(buf \texttt{ >>= } f) \texttt{ in}\\ \langle F\ \langle X', \texttt{if bnull}(b') \texttt{ then } buf \texttt{ else } b'\rangle, b'\rangle\end{array}\right))\ \langle X, init\rangle$$

The state maintained by $\texttt{buffer}(X, init)$ is the pair of the state maintained by X and its previous outputs (initially *init*).

5.2 Example

The following process uses the three combinators presented above:

$$P = \texttt{buffer}((\lambda n.\lambda buf.\{n + buf\}) \texttt{ o } \texttt{base}(\lambda m.\{m\}), \{0\})$$

This process maintains a state constituted of a single integer, initialized to 0. Its inputs are integers. At any point in time, its state is the sum of all the inputs it has received in the past. Because the combinators used in P are defined as fixpoints, P contains three nested occurrences of `fix`. We will now show that P is computationally equivalent to the following even simpler process:

$$P' = \texttt{fix}(\lambda F.\lambda s.\lambda m.\texttt{let } x ::= m + s \texttt{ in } \langle F\ x, \{x\}\rangle)\ 0$$

Using Nuprl's powerful tactic mechanism we automatically generate P' from P, and we automatically prove that $P \sim P'$. Our experiments showed that it takes between 100 and 200 computation steps for P to process a single input while it takes less than 10 computation steps for P' to process a single input.

Standard Form To optimize our processes we take advantage of the fact that many of them are of the following form:

$$\texttt{process}(n, L, S, R, I) =$$
$$\texttt{fix}(\lambda F.\lambda s.\lambda m.\left(\begin{array}{l}\texttt{let } x_1 ::= L\ s\ m\ 1 \texttt{ in}\\ \cdots\\ \texttt{let } x_n ::= L\ s\ m\ n\ x_1\ \cdots\ x_{n-1} \texttt{ in}\\ \langle F\ (S\ s\ m\ x_1\ \cdots\ x_n), R\ s\ m\ x_1\ \cdots\ x_n\rangle\end{array}\right))\ I$$

where L is a sequence of instructions defined as a function, n is the number of instructions that the process executes on each input, S computes the next state of the process, R computes the outputs, and I is its initial state.

Transformations. We prove the next three lemmas using the same proof technique as in Sec. 4.7. These lemmas show that if processes are built using the base, composition, and buffer combinators (and many other primitive combinators of the GPM not presented in this paper), then they are guaranteed to be of the standard form $\texttt{process}(n, L, S, R, I)$.

Lemma 10. *Given a term f of type* Top*, the following bisimulation is true:*

$$\texttt{base}(f) \sim \texttt{process}(0, \lambda x.\bot, \lambda s.\lambda m.\texttt{Ax}, \lambda s.\lambda m.f\ m, \texttt{Ax})$$

Lemma 11. *Given f, L, S, R, and I terms of type* Top*, and n a natural number, the following bisimulation is true:*

$$\begin{array}{l} f\ \texttt{o}\ \texttt{process}(n, L, S, R, I) \\ \sim \texttt{process}(\ n+1, \\ \qquad \lambda s.\lambda m.\lambda i.\ \texttt{if}\ i = n+1\ \texttt{then}\ \lambda x_1 \ldots . \lambda x_n.\ \texttt{bmap}(f, R\ s\ m\ x_1\ \cdots\ x_n), \\ \qquad\qquad\qquad \texttt{else}\ L\ s\ m \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.S\ s\ m\ x_1\ \cdots\ x_n, \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.x_{n+1}, \\ \qquad I) \end{array}$$

Lemma 12. *Given L, S, R, I, and I' terms of type* Top*, and n a natural number, the following bisimulation is true:*

$$\begin{array}{l} \texttt{buffer}(\texttt{process}(n, L, S, R, I), I') \\ \sim \texttt{process}(\ n+1, \\ \qquad \lambda s.\lambda m.\lambda i.\ \texttt{if}\ i = n+1 \\ \qquad\qquad\qquad \texttt{then}\ \lambda x_1 \ldots . \lambda x_n.\ (R\ \pi_1(s)\ m\ x_1\ \cdots\ x_n)\ \texttt{>>=}\ \lambda f.(\pi_2(s)\ \texttt{>>=}\ f) \\ \qquad\qquad\qquad \texttt{else}\ L\ \pi_1(s)\ m \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.\langle S\ \pi_1(s)\ m\ x_1\ \cdots\ x_n \\ \qquad\qquad\qquad\qquad , \texttt{if bnull}(x_{n+1})\ \texttt{then}\ s\ \texttt{else}\ x_{n+1}\rangle \\ \qquad \lambda s.\lambda m.\lambda x_1 \ldots . \lambda x_n.\ \lambda x_{n+1}.x_{n+1}, \\ \qquad \langle I, I'\rangle) \end{array}$$

Transformation of P into P'. Using the bisimulations presented above, we automatically rewrite P into P', and because our bisimulations are untyped, proving that P is computationally equivalent to P' is also trivial: it only requires us to prove that some terms are in Top, and all closed terms are trivially in Top.

6 Related Work and Conclusion

This paper describes computational proof techniques based on bisimulations which we use in the Nuprl proof assistant in order to optimize distributed processes (programs in general). McCarthy [31] recognized the value of type free

reasoning, and we took that to heart in the design of CTT by providing type free rules about computation, called "direct computation rules". Now we know that this kind of reasoning can be made even richer.

Gordon [20] characterizes contextual equivalence as some form of co-inductively defined bisimulation. Using co-inductive reasoning, Gordon can easily prove, e.g., various bisimulations between streams. For example, he proves that $\texttt{iterate}(f, f\ x)$ and $\texttt{map}(f, \texttt{iterate}(f, x))$ are bisimilar, where $\texttt{iterate}(f, x)$ is defined in Nuprl as $\texttt{fix}(\lambda F.\lambda x.\langle x, F\ (f\ x)\rangle)\ x$. Nuprl's corresponding method to prove such results is the least upper bound property. Gordon proves this result using a co-inductive reasoning, while we prove it by induction on the natural number we obtain by approximating the two fixpoints used to define `map` and `iterate`. Apart from that difference, the resulting proofs are similar in spirit.

Note that we have not yet formally proved that the processes returned by our optimizer have a better complexity than the processes it takes as inputs. Using Isabelle/HOL, Aspinall, Beringer, and Momigliano [5] developed an optimization validation technique, based on a proof-carrying code approach, to prove that optimized programs use less resources than the non-optimized versions. Currently, we cannot measure the complexity of programs inside Nuprl because if t_1 reduces to t_2 then $t_1 \sim t_2$, and hence we cannot distinguish between them in any context.

We hope to solve this issue by either using some kind of reflection, or introducing a subtype of `Base` where equality would be alpha-equality. Also, in order to enhance the usability of our processes in industrial strength systems, we need to identify and verify other optimizations. As mentioned in Sec. 1, we view this work as a step towards making Nuprl a usable programming framework. In the meantime, we have built a Lisp translator for our processes.

In the last two decades, much work has been done on compiler verification. See Dave [19] for earlier references. To name a few: Using Coq, Leroy has developed and certified a compiler for a C-like language [29]. He generated the compiler using Coq's extraction mechanism to Caml code. The compiler is certified thanks to "a machine-checked proof of semantic preservation" [29]. Also using Coq, Chlipala [15] developed a verified compiler for an impure functional programming language with references and exceptions that produces code in an idealized assembly language. He proved the correctness of the compiler using a big-step operational semantics. Li [30] designed a verified compiler in HOL, from an high-level ML-like programming language implemented in HOL to ARM assembly code. Each transformation of the compiler generates a correctness argument along with a piece of code.

Following this line of work, we now would like to tackle the task of building a verified compiler for Nuprl in Nuprl.

Acknowledgements. We would like to thank our colleagues Professor Robert L. Constable, David Guaspari, and Evan Moran for their helpful criticism.

References

1. The Coq Proof Assistant, http://coq.inria.fr/
2. Allen, S.F.: A non-type-theoretic definition of martin-löf's types. In: LICS, pp. 215–221. IEEE Computer Society (1987)
3. Allen, S.F.: A Non-Type-Theoretic Semantics for Type-Theoretic Language. PhD thesis, Cornell University (1987)
4. Allen, S.F., Bickford, M., Constable, R.L., Eaton, R., Kreitz, C., Lorigo, L., Moran, E.: Innovations in computational type theory using Nuprl. J. Applied Logic 4(4), 428–469 (2006)
5. Aspinall, D., Beringer, L., Momigliano, A.: Optimisation validation. Electr. Notes Theor. Comput. Sci. 176(3), 37–59 (2007)
6. Barzilay, E.: Implementing Reflection in NUPRL. PhD thesis, Cornell University (2006)
7. Berghofer, S.: Program extraction in simply-typed higher order logic. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 21–38. Springer, Heidelberg (2003)
8. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 24–40. Springer, Heidelberg (2002)
9. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
10. Bickford, M.: Component specification using event classes. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 140–155. Springer, Heidelberg (2009)
11. Bickford, M., Constable, R., Guaspari, D.: Generating event logics with higher-order processes as realizers. Technical report, Cornell University (2010)
12. Bickford, M., Constable, R.L.: Formal foundations of computer security. In: NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 14, pp. 29–52 (2008)
13. Bickford, M., Constable, R.L., Rahli, V.: Logic of events, a framework to reason about distributed systems. In: Languages for Distributed Algorithms Workshop (2012)
14. Charron-Bost, B., Schiper, A.: The Heard-Of model: Computing in distributed systems with benign failures. Distributed Computing 22(1), 49–71 (2009)
15. Chlipala, A.: A verified compiler for an impure functional language. In: 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 93–106. ACM (2010)
16. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing mathematics with the Nuprl proof development system. Prentice-Hall, Inc., Upper Saddle River (1986)
17. Constable, R.L., Smith, S.F.: Computational foundations of basic recursive function theory. Theoretical Computer Science 121(1&2), 89–112 (1993)
18. Crary, K.: Type-Theoretic Methodology for Practical Programming Languages. PhD thesis, Cornell University, Ithaca, NY (August 1998)
19. Dave, M.A.: Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 28(6), 2 (2003)
20. Gordon, A.D.: Bisimilarity as a theory of functional programming. Electr. Notes Theor. Comput. Sci. 1, 232–252 (1995)

21. Hickey, J., et al.: MetaPRL - A modular logical environment. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 287–303. Springer, Heidelberg (2003)
22. Hickey, J.J.: The MetaPRL Logical Programming Environment. PhD thesis, Cornell University, Ithaca, NY (January 2001)
23. Howe, D.J.: Equality in lazy computation systems. In: Proceedings of Fourth IEEE Symposium on Logic in Computer Science, pp. 198–203. IEEE Computer Society (1989)
24. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. 124(2), 103–112 (1996)
25. Kopylov, A.: Dependent intersection: A new way of defining records in type theory. In: LICS, pp. 86–95. IEEE Computer Society (2003)
26. Kopylov, A.: Type Theoretical Foundations for Data Structures, Classes, and Objects. PhD thesis, Cornell University, Ithaca, NY (2004)
27. Kreitz, C.: The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide. Cornell University, Ithaca, NY (2002), `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`
28. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
29. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In: 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 42–54. ACM (2006)
30. Li, G.: Formal Verification of Programs and Their Transformations. PhD thesis, University of Utah (2010)
31. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, Part I. Commun. ACM 3(4), 184–195 (1960)
32. Mendler, P.F.: Inductive Definition in Type Theory. PhD thesis, Cornell University, Ithaca, NY (1988)
33. Rahli, V., Schiper, N., Renesse, R.V., Bickford, M., Constable, R.L.: A diversified and correct-by-construction broadcast service. In: The 2nd Int'l Workshop on Rigorous Protocol Engineering (WRiPE), Austin, TX (October 2012)
34. Schiper, N., Rahli, V., Van Renesse, R., Bickford, M., Constable, R.L.: ShadowDB: A replicated database on a synthesized consensus core. In: Eighth Workshop on Hot Topics in System Dependability, HotDep 2012 (2012)
35. Smith, S.F.: Partial Objects in Type Theory. PhD thesis, Cornell University, Ithaca, NY (1989)

Type Classes and Filters for Mathematical Analysis in Isabelle/HOL

Johannes Hölzl[1,⋆], Fabian Immler[1,⋆⋆], and Brian Huffman[2]

[1] Institut für Informatik, Technische Universität München
{hoelzl,immler}@in.tum.de
[2] Galois, Inc.
huffman@galois.com

Abstract. The theory of analysis in Isabelle/HOL derives from earlier formalizations that were limited to specific concrete types: $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{R}^n$. Isabelle's new analysis theory unifies and generalizes these earlier efforts. The improvements are centered on two primary contributions: a generic theory of limits based on filters, and a new hierarchy of type classes that includes various topological, metric, vector, and algebraic spaces. These let us apply many results in multivariate analysis to types which are not Euclidean spaces, such as the extended real numbers, bounded continuous functions, or finite maps.

Keywords: Type classes, Filters, Mathematical analysis, Topology, Limits, Euclidean vector spaces, Isabelle/HOL.

1 Introduction

Mathematical analysis studies a hierarchy of abstract objects, including various topological, metric, and vector spaces. However, previous formalizations of mathematical analysis have not captured this hierarchical structure. For example, in HOL Light's multivariate analysis library [4] most theorems are proved only for the fixed type $\mathbb{R}^n$ of finite Cartesian products. Similarly, Isabelle's original library of analysis by Fleuriot and Paulson [1] supported most concepts only on $\mathbb{R}$ and $\mathbb{C}$.

Isabelle/HOL's new library for mathematical analysis derives from these two earlier libraries, but brings them closer to the mathematical ideal: Isabelle/HOL provides the concept of type classes, which allows us to state lemmas generically for all types that provide the necessary operations and satisfy the corresponding assumptions. This approach is therefore perfectly suited to exhibit the hierarchical structure of spaces within mathematical analysis.

In the following text we present the new hierarchy of type classes for mathematical analysis in Isabelle/HOL and preview some example class instances:

- Finite Cartesian products $\mathbb{R}^n$, $\mathbb{R}$, and $\mathbb{C}$ are all Euclidean spaces.
- The extended reals $\overline{\mathbb{R}} = \mathbb{R} \cup \{\infty, -\infty\}$ are a non-metric topological space.

⋆ Supported by the DFG Projekt NI 491/15-1.
⋆⋆ Supported by the DFG Graduiertenkolleg 1480 (PUMA).

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 279–294, 2013.

– Finite maps (maps with finite domain) $\mathbb{N} \to_f \mathbb{R}$, are a complete metric space but not a vector space. They are used to construct stochastic processes [6].
– Bounded continuous functions $\mathbb{R} \to_{bc} \mathbb{R}$ form a Banach space but not a Euclidean space; their dimension is infinite. They are used to prove that ordinary differential equations have a unique solution [7].

Figure 1 shows the type class hierarchy we present in this paper. Full lines are inheritance relations and dashed lines are proved subclass relations. We group the type classes into topological, metric, vector and algebraic type classes. For completeness we show some of the algebraic type classes, but they are not the main focus of this paper. All type classes described in this paper are available in Isabelle 2013 and carry the same names in the formalization. An exception is the order topology, available in Isabelle's development repository[1].

Our formalization of filters and limits is another primary contribution of our work. While filters have long been used to express limits in topology, our generic limit operator parameterized by two filters is novel (see Section 4.2). Filters are also useful for more than just limits—e.g. a filter can express the *almost everywhere* quantifier, which recognizes predicates that hold with probability 1 on a probability space.

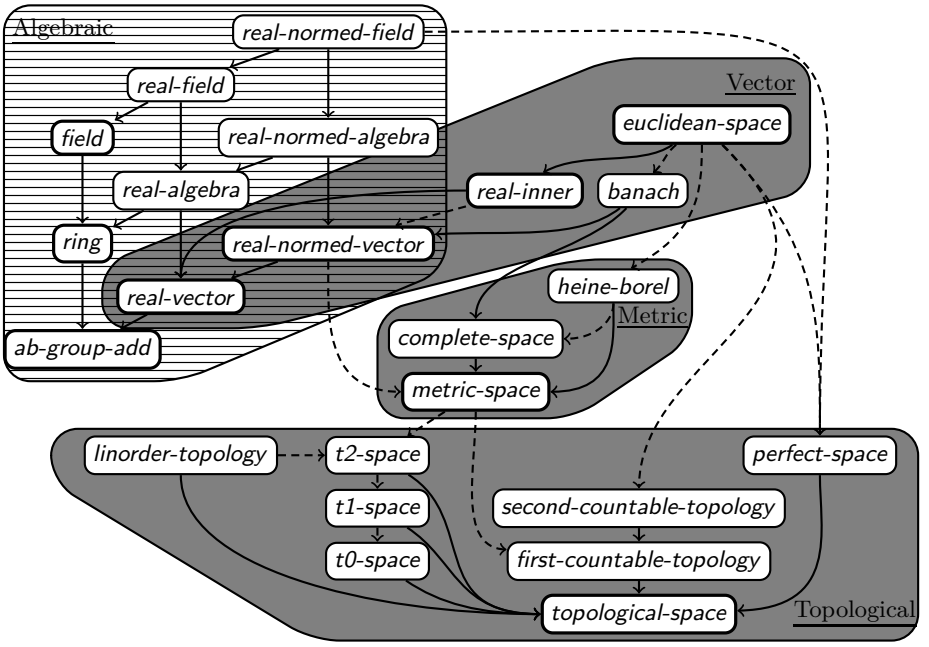


Fig. 1. Type class hierarchy

[1] `http://isabelle.in.tum.de/repos/isabelle/rev/4392eb046a97`

2 Preliminaries

The term syntax used in this paper follows Isabelle/HOL, i.e. as usual in λ-calculus function application is juxtaposition as in $f\ t$. The notation $t :: \tau$ means that term t has type τ. Types are built from the base types $\mathbb{B}$ (booleans), $\mathbb{N}$ (natural numbers), $\mathbb{R}$ (reals), type variables (α, β, etc), via the function type constructor $\alpha \to \beta$, and via the set type constructor $\alpha\ \mathsf{set}$. The universe of a type α, i.e. the set of all elements of type α, is written $\mathcal{U}_\alpha$. We use $\Longrightarrow$ for logical implication; it binds – in contrast to Isabelle/HOL notation – stronger than universal quantification, i.e. $\forall x.\ P\ x \Longrightarrow Q\ x$ equals $\forall x.\ (P\ x \Longrightarrow Q\ x)$.

Isabelle/HOL provides (axiomatic) type classes [2], which allow to organize polymorphic specifications. A type class C specifies assumptions $P_1, \ldots, P_k$ for constants $c_1, \ldots, c_m$ (that are to be overloaded) and may be based on other type classes $B_1, \ldots, B_n$. The command **class** declares type classes in Isabelle/HOL:

class $C = B_1 + B_2 + \ldots + B_n +$
  fixes $c_1 :: \alpha\ \kappa_1$ **and** $c_2 :: \alpha\ \kappa_2$ **and** $\cdots$ **and** $c_m :: \alpha\ \kappa_m$
  assumes P_1 **and** P_2 **and** $\cdots$ **and** P_k

In the type class specification only one type variable, α, is allowed to occur. Variables in $P_1, \ldots, P_k$ are implicitly universally quantified. A type α is said to be an instance of the type class C if it provides definitions for the respective constants and respects the required assumptions. In this case we write $\alpha :: C$.

With the command **instance** we can add subclass relations in addition to the declared base classes. We have for example the type class *finite* for types α where $\mathcal{U}_\alpha$ is finite and a type class *countable* for types α where $\mathcal{U}_\alpha$ is countable. Then we can use **instance** $\mathit{finite} \subseteq \mathit{countable}$ to add a subclass relation stating that all finite types are also countable types.

3 Related Work

Isabelle's original theory of real analysis was due to Fleuriot and Paulson [1]. It covered sequences, series, limits, continuity, transcendental functions, nth roots, and derivatives. These notions were all specific to $\mathbb{R}$, although much was also duplicated at type $\mathbb{C}$. This material has since been adapted to the new type class hierarchy. The non-standard analysis part with ${}^*\mathbb{R}$ and ${}^*\mathbb{C}$ is not adapted.

Much of the work presented in this paper comes from the Isabelle/HOL port of Harrison's multivariate analysis library for HOL Light [4]. In addition to limits, convergence, continuity, and derivatives, the library also covers topology and linear algebra. The Heinstock-Kurzweil integral is not yet described in [4], but it is now available in HOL Light and also ported to Isabelle/HOL. Compared to the work presented in this paper the HOL Light library is mostly specific to $\mathbb{R}^n$.

Instead of formalizing limits with filters, Harrison invented a variant of nets which also bore some similarities to filter bases. His library provided a tends-to relation parameterized by a single net, but did not have an equivalent of our more general limit operator which is parameterized by two filters (see Section 4.2).

Lester [9] uses PVS to formalize topology. He formalizes topological spaces, T_2-spaces, second countable space, and metric spaces. He does not provide vector spaces *above* metric spaces and he does not use filters or nets to express limits.

Spitters and van der Weegen [10] formalize a type class hierarchy for algebraic types in Coq. Their goal is efficient computation, hence they support different implementations for isomorphic types. In contrast, our goal is to share definitions and proofs for types which share the same mathematical structure. They also introduce type classes in category theory which is not possible in Isabelle as type classes are restricted to one type variable. However, for mathematical analysis and also for the algebraic type class hierarchy in Isabelle/HOL they suffice.

Hölzl and Heller [5], Immler and Hölzl [7], and Immler [6] provide instances of the type classes presented in this paper: they formalize extended real numbers, bounded continuous functions, and finite maps.

4 Topology

Topology is concerned with expressing *nearness* of elements in a space. An open set contains for each element also all elements which are in some sense *near* it. This structure is sufficient to express limits and continuity of functions on topological spaces. This generality is actually needed to formulate a notion of limits and convergence that is also suitable for extended real numbers. More specific formulations (e.g. in terms of metric spaces) do not work for them. For a introduction into topology the reader may look into standard textbooks like [8].

4.1 Topological Spaces

A *topological space* is defined by its predicate of open sets. In mathematics the support space X, the union of all open sets, is usually explicitly given. In Isabelle/HOL a topological space is a type in the following type class:

class *topological-space* =
  fixes *open* :: $\alpha\ \mathit{set} \to \mathbb{B}$
  assumes *open* $\mathcal{U}_\alpha$ **and** *open* $U \implies$ *open* $V \implies$ *open* $(U \cap V)$
    and $(\forall U \in S.\ \mathit{open}\ U) \implies \mathit{open} \bigcup S$
closed :: $\alpha\ \mathit{set} \to \mathbb{B}$
closed $U \iff$ *open* $(\mathcal{U}_\alpha \setminus U)$

On a topological space, we define the limit points, interior, closure and frontier of a set in the usual way.

On the real numbers, the canonical topology contains all half-bounded open intervals: $]a, \infty[$ and $]\infty, a[$. It is also generated by them, i.e. it is the smallest topology containing all half-bounded open intervals. This is called an *order topology* on a linear order:

class *linorder-topology* = *linorder* + *topological-space* +
  assumes *open* = *generated-topology* $\bigcup_x \{]x, \infty[,]\infty, x[\}$

Here *generated-topology* A is the smallest topology where the sets in A are open. A is a subbase, i.e. A need not be closed under intersection. Instances for order topologies are the real numbers and the extended real numbers.

Separation Spaces. As the open sets of a topology describe only *nearness*, it is still possible that two distinct elements are always near, i.e. they are *topologically indistinguishable*. This is not desirable when formulating unique limits in terms of open sets. To prevent this, different classes of *separation spaces* are specified, called T_0-, T_1-, and T_2-*spaces*:

class *t0-space* = *topological-space* +
  assumes $x \neq y \implies \exists U.\ \mathit{open}\ U \wedge \big(x \in U \iff y \notin U\big)$

class *t1-space* = *topological-space* +
  assumes $x \neq y \implies \exists U.\ \mathit{open}\ U \wedge \big(x \in U \wedge y \notin U\big)$

In T_1-spaces singleton sets are closed, i.e. *closed* $\{x\}$. A T_2-space (also called a Hausdorff space) is the strongest separation space we provide. A T_2-space provides for any distinct elements x and y two disjoint open sets around them:

class *t2-space* = *topological-space* +
  assumes $x \neq y \implies$
    $\exists U, V.\ \mathit{open}\ U \wedge \mathit{open}\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \emptyset$

We provide type class inclusion for these spaces according to their numbering; i.e. a T_2-space is also a T_1-space is also a T_0-space. In Section 5.1 we also prove that each metric space and each linearly ordered topology is a T_2-space.

instance *t1-space* $\subseteq$ *t0-space*
instance *t2-space* $\subseteq$ *t1-space*
instance *linorder-topology* $\subseteq$ *t2-space*

While the T_1-spaces tell us that two elements can always be separated, we also need its dual: in a *perfect space* each open set containing an element always contains elements around it; the singleton set is never open. This is the dual to *closed* $\{a\}$. Only in perfect spaces is $\lim_{x \to a}$ meaningful for each point a.

class *perfect-space* = *topological-space* + **assumes** $\neg \mathit{open}\ \{a\}$

Instances of perfect spaces are Euclidean spaces and the extended real numbers.

Topologies with Countable Basis. A *first countable topology* assumes a countable basis for the neighborhoods of every point; i.e. it allows us to construct a sequence of open sets that converges towards a point x. Together with a T_1-space this allows us to construct a sequence of points that converges to a point x.

class *first-countable-topology* = *topological-space* +
  assumes $\exists A.\ \mathit{countable}\ A \wedge (\forall a \in A.\ \mathit{open}\ a \wedge x \in a) \wedge$
    $(\forall U.\ \mathit{open}\ U \wedge x \in U \implies \exists a \in A.\ a \subseteq U)$

Examples of first countable topologies are metric spaces.

Second countability is an extension of first countability; it provides a countable basis for the whole topology, not just for the neighborhoods of every point. This implies that compactness is equivalent to sequential compactness (which will be introduced in Section 4.4).

class *second-countable-topology* = *topological-space* +
  assumes $\exists B.$ *countable* $B \wedge$ *open* = *generated-topology* B
instance *second-countable-topology* $\subseteq$ *first-countable-topology*

Instances for second countable spaces are Euclidean spaces, the extended real numbers, and finite maps $(\alpha ::$ *countable*$) \rightharpoonup_f (\beta ::$ *second-countable-topology*$)$.

4.2 Filters and Limits

A *filter* is a set of sets (or equivalently a predicate on predicates) with a certain order structure. As we will see shortly, filters are useful in topology because they let us unify various kinds of limits and convergence, including limits of sequences, limits of functions at a point, one-sided and asymptotic limits.

Many varieties of logical quantification are filters, such as "for all x in set A"; "for sufficiently large n"; "for all but finitely many x"; "for x sufficiently close to y". These quantifiers are similar to the ordinary universal quantifier ($\forall$) in many ways. In particular, each holds for the always-true predicate, preserves conjunction, and is monotonic:

$(\square x.\ \mathit{True})$
$(\square x.\ P\ x) \implies (\square x.\ Q\ x) \implies (\square x.\ P\ x \wedge Q\ x)$
$(\forall x.\ P\ x \implies Q\ x) \implies (\square x.\ P\ x) \implies (\square x.\ Q\ x)$

We define a filter $\mathcal{F}$ as a predicate on predicates that satisfies all three of the above rules. (Note that we do not require filters to be *proper*; that is, we admit the trivial filter "for all x in $\{\}$" which holds for all predicates, including $\lambda x.\ \mathit{False}$.)

is-filter $:: ((\alpha \to \mathbb{B}) \to \mathbb{B}) \to \mathbb{B}$
is-filter $\mathcal{F} =$
  $\mathcal{F}\ (\lambda x.\mathit{True}) \wedge$
  $(\forall P, Q.\ \mathcal{F}\ (\lambda x.\ P\ x) \implies \mathcal{F}\ (\lambda x.\ Q\ x) \implies \mathcal{F}\ (\lambda x.\ P\ x \wedge Q\ x)) \wedge$
  $(\forall P, Q.\ (\forall x.\ P\ x \implies Q\ x) \implies \mathcal{F}\ (\lambda x.\ P\ x) \implies \mathcal{F}\ (\lambda x.\ Q\ x))$

We define the type α *filter* comprising all filters over the type α. The command **typedef** provides functions $\mathit{Rep}_{\mathit{filter}}$ and $\mathit{Abs}_{\mathit{filter}}$ to convert between α *filter* and $(\alpha \to \mathbb{B}) \to \mathbb{B}$; we use *eventually* $:: (\alpha \to \mathbb{B}) \to \alpha$ *filter* $\to \mathbb{B}$ (defined as $\mathit{Rep}_{\mathit{filter}}$ with swapped argument order) to apply a filter to a predicate.

typedef α *filter* $= \{\mathcal{F} \mid$ *is-filter* $\mathcal{F}\}$

Note: For each filter $F :: \alpha$ *filter*, we will usually show only its characteristic equation *eventually* $P\ F \iff \mathcal{F}\ P$, leaving the raw definition $F = \mathit{Abs}_{\mathit{filter}}\ \mathcal{F}$ and the proof obligation *is-filter* $\mathcal{F}$ implicit.

Finer-than Ordering. We define the ordering $F_1 \leq F_2$ to mean that filter F_1 is *finer than* F_2, i.e., $\forall P.\ \mathsf{eventually}\ P\ F_2 \implies \mathsf{eventually}\ P\ F_1$. For filters that represent bounded quantifiers, $\leq$ agrees with the subset order: "for all x in A" $\leq$ "for all x in B" iff $A \subseteq B$. This ordering also makes $\alpha\ \mathsf{filter}$ into a complete lattice, with the trivial filter as the bottom element and $\forall$ as the top element.

$\square \leq \square \quad :: \quad \alpha\ \mathsf{filter} \to \alpha\ \mathsf{filter} \to \mathbb{B}$
$F_1 \leq F_2 \longleftrightarrow (\forall P.\ \mathsf{eventually}\ P\ F_2 \implies \mathsf{eventually}\ P\ F_1)$

Basic Filters. On any linearly ordered type, we define filters *at-top* to mean "for sufficiently large y" or "as $y \longrightarrow +\infty$", and *at-bot* as "for sufficiently small y" or "as $y \longrightarrow -\infty$". We use *sequentially* as an abbreviation for *at-top* as a filter on the naturals.

lemma
$\mathsf{eventually}\ P\ (\mathsf{at\text{-}top} :: (\alpha :: \mathsf{linorder})\ \mathsf{filter}) \longleftrightarrow (\exists x.\ \forall y \geq x.\ P\ y)$
$\mathsf{eventually}\ P\ (\mathsf{at\text{-}bot} :: (\alpha :: \mathsf{linorder})\ \mathsf{filter}) \longleftrightarrow (\exists x.\ \forall y \leq x.\ P\ y)$

In the context of a topological space, we define *nhds* x as the *neighborhood filter*, which means "for all y in some open neighborhood of x".

lemma
$\mathsf{eventually}\ P\ (\mathsf{nhds}\ x) \longleftrightarrow (\exists U.\ \mathsf{open}\ x \wedge x \in U \wedge (\forall y \in U.\ P\ y))$

The *principal filter* of a set B represent a bounded quantifier, i.e. "for all x in B". It is useful for constructing refinements of the neighborhood filter. We define $\mathsf{at}\ x\ \mathsf{within}\ U$ as the *punctured* neighborhood filter, "for all $y \in U$ and $y \neq x$ in some neighborhood of x". We also define one-sided filters *at-left* and *at-right*. $\mathsf{at}\ x$ is an abbreviation for $\mathsf{at}\ x\ \mathsf{within}\ \mathcal{U}_\alpha$. $F_1 \sqcap F_2$ is the infimum of the filters F_1 and F_2.

lemma
$\mathsf{eventually}\ P\ (\mathsf{principal}\ S) \longleftrightarrow (\forall x \in S.\ P\ x)$

$\mathsf{at}\ \square\ \mathsf{within}\ \square\ :: (\alpha :: \mathsf{topological\text{-}space}) \to \alpha\ \mathsf{set} \to \alpha\ \mathsf{filter}$
$\mathsf{at\text{-}left}, \mathsf{at\text{-}right} :: (\alpha :: \mathsf{linorder\text{-}topology}) \to \alpha\ \mathsf{filter}$

$\mathsf{at}\ x\ \mathsf{within}\ U = \mathsf{nhds}\ x \sqcap \mathsf{principal}\ (U \setminus \{x\})$
$\mathsf{at\text{-}left}\ x = \mathsf{at}\ x\ \mathsf{within}\ ]\infty, x[$
$\mathsf{at\text{-}right}\ x = \mathsf{at}\ x\ \mathsf{within}\ ]x, \infty[$

When we apply a function to the argument of each predicate in a filter we get a filter again. With $\mathsf{filtermap}\ f\ F$ we transform the filter F by a function f. We will shortly use it for expressing general limits.

lemma
$\mathsf{eventually}\ P\ (\mathsf{filtermap}\ f\ F) \longleftrightarrow \mathsf{eventually}\ (\lambda x.\ P\ (f\ x))\ F$

lemma
$\mathsf{filtermap}\ (\lambda x :: \mathbb{R}.\ -x)\ \mathsf{at\text{-}top} = \mathsf{at\text{-}bot}$
$\mathsf{filtermap}\ (\lambda x :: \mathbb{R}.\ 1/x)\ \mathsf{at\text{-}top} = \mathsf{at\text{-}right}\ 0$

Limits. Filters can be used to express a general notion of limits. To illustrate this, we start with the usual epsilon-delta definitions of limits of functions and sequences on reals, and then incrementally generalize the definitions. Finally we end up with a single definition, parameterized over two filters, that can express diverse kinds of limits in arbitrary topological spaces. Here are the usual epsilon-delta definitions of limits for sequences and for functions at a point.

$$(y_n \longrightarrow L) = (\forall \epsilon > 0.\ \exists n_0.\ \forall n \geq n_0.\ |y_n - L| < \epsilon)$$
$$(\lim_{x \to a} f(x) = L) = (\forall \epsilon > 0.\ \exists \delta > 0.\ \forall x.\ 0 < |x - a| < \delta \implies |f(x) - L| < \epsilon)$$

The reader may recognize "$\exists n_0.\ \forall n \geq n_0$" as the filter *sequentially*. Also note that "$\exists \delta > 0.\ \forall x.\ 0 < |x - a| < \delta$" is equivalent to the punctured neighborhood filter (*at* a). Therefore we can rewrite the above definitions as follows.

$$(y_n \longrightarrow L) = (\forall \epsilon > 0.\ \textit{eventually}\ (\lambda n.\ |y_n - L| < \epsilon)\ \textit{sequentially})$$
$$(\lim_{x \to a} f(x) = L) = (\forall \epsilon > 0.\ \textit{eventually}\ (\lambda x.\ |f(x) - L| < \epsilon)\ (\textit{at}\ a))$$

Already we can unify these two definitions by parameterizing over the filter. (This yields the same definition as the *tendsto* relation from HOL Light.)

$$(f \longrightarrow L)\ F = (\forall \epsilon > 0.\ \textit{eventually}\ (\lambda x.\ |f(x) - L| < \epsilon)\ F) \tag{1}$$

We express many kinds of limits with $(f \longrightarrow x)\ F$ by instantiating F with various filters: *sequentially* for sequences, *at* a for a function at a point, *at-top* or *at-bot* for a function at $\pm\infty$, *at-left* a or *at-right* a for one-sided limits.

$$(x_n \longrightarrow L) = (x \longrightarrow L)\ \textit{sequentially}$$
$$(\lim_{x \to a} f(x) = L) = (f \longrightarrow L)\ (\textit{at}\ a)$$
$$(\lim_{x \to a^+} f(x) = L) = (f \longrightarrow L)\ (\textit{at-right}\ a)$$
$$(\lim_{x \to -\infty} f(x) = L) = (f \longrightarrow L)\ \textit{at-bot}$$

Up to now, we generalized how the limit is approached, but we can also generalize the right-hand side L. First we rewrite (1) using *filtermap*:

$$(f \longrightarrow L)\ F = (\forall \epsilon > 0.\ \textit{eventually}\ (\lambda y.\ |y - L| < \epsilon)\ (\textit{filtermap}\ f\ F))$$

This says that *filtermap* f F is eventually in every open neighborhood of L, which is equivalent to the following:

$$(f \longrightarrow L)\ F = (\textit{filtermap}\ f\ F \leq \textit{nhds}\ L)$$

Finally, we can generalize *nhds* L to an arbitrary filter G and obtain the generalized limit *LIM* x *in* F. $f\ x :> G$ (in Isabelle/HOL also written *filterlim* f F G).

$$\begin{aligned}
&\textit{LIM}\ \square\ \textit{in}\ \square.\ \square :> \square && :: (\alpha \to \beta) \to \alpha\ \textit{filter} \to \beta\ \textit{filter} \to \mathbb{B} \\
&\textit{LIM}\ x\ \textit{in}\ F.\ f\ x :> G && \iff \textit{filtermap}\ f\ F \leq G \\
&(\square \longrightarrow \square)\ \square && :: (\alpha \to \beta) \to (\beta :: \textit{topological-space}) \to \alpha\ \textit{filter} \to \mathbb{B} \\
&(f \longrightarrow L)\ F && \iff \textit{LIM}\ x\ \textit{in}\ F.\ f\ x :> \textit{nhds}\ L \\
&\square \longrightarrow \square && :: (\mathbb{N} \to \alpha) \to (\alpha :: \textit{topological-space}) \to \mathbb{B} \\
&X \longrightarrow L && \iff (X \longrightarrow L)\ \textit{sequentially}
\end{aligned}$$

This abstract notion of limit is only based on filters and does not even require topologies. Now we can express new limits (that are not expressible in HOL Light's library), e.g., *LIM x in at-bot.* $-x$:> *at-top* says that $-x$ goes to infinity as x approaches negative infinity, $\lim_{x \to -\infty} -x = \infty$.

For *filterlim* we can provide a composition rule for convergence. Further rules about e.g. elementary functions are available for normed vector spaces.

lemma
$(LIM\ x\ in\ F_1.\ f\ x :> F_2) \Longrightarrow (LIM\ x\ in\ F_2.\ g\ x :> F_3) \Longrightarrow$
$(LIM\ x\ in\ F_1.\ g\ (f\ x) :> F_3)$

We can prove e.g. $((\lambda x.\ exp\ (-1/x)) \longrightarrow 0)\ (at\text{-}right\ 0)$ from $(exp \longrightarrow 0)$ *at-bot*, *LIM x in at-top.* $-x$:> *at-bot*, and *LIM x in at-right* $0.\ 1/x$:> *at-top*.

On the order topology, a function converges to x iff for all upper and lower bounds of x the function is eventually in these bounds.

lemma fixes $f :: \alpha \to (\beta :: linorder\text{-}topology)$
shows $(f \longrightarrow x)\ F \Longleftrightarrow (\forall b > x.\ eventually\ (\lambda x.\ f\ x < b)\ F) \wedge$
$(\forall b < x.\ eventually\ (\lambda x.\ b < f\ x)\ F)$

Filters vs nets. As an alternative to filters, limits may also be defined using *nets*, which generalize sequences. While sequences are indexed by natural numbers, a net may be indexed by any directed set. Like filters, nets support an "eventually" operator: N eventually satisfies P iff $\exists x.\ \forall y \geq x.\ P(N(y))$.

In terms of formalizing limits and convergence, filters and nets are equally expressive. However, nets are not as convenient to formalize in HOL. A type α *net* of all nets over α does not work; nets require a second parameter type to allow arbitrary index sets.

4.3 Continuity

Continuity of a function f at a filter F says that the function converges on F towards its value $f\ x$ where F converges to x. We use filters to unify continuity at a point, continuity from left, continuity from right etc. With *Lim* $F\ (\lambda x.\ x)$ we select the convergence point of the filter F with definite choice. To have a unique value for x, the domain of the function needs to be a T_2-space.

$Lim :: \alpha\ filter \to (\alpha \to \beta) \to (\beta :: t2\text{-}space)$
$Lim\ F\ f = THE\ L.\ (f \longrightarrow L)\ F$

$continuous :: \alpha\ filter \to (\alpha :: t2\text{-}space \to \beta :: topological\text{-}space) \to \mathbb{B}$
$continuous\ F\ f \Longleftrightarrow (f \longrightarrow f\ (Lim\ F\ (\lambda x.\ x)))\ F$

This is similar to the definition in HOL Light, but generalized to topological spaces instead of Euclidean spaces.

Often a function needs to be continuous not only at a point, but on a set. For this we introduce *continuous-on*. Its domain is not restricted to a T_2-space.

$continuous\text{-}on :: \alpha\ set \to (\alpha :: topological\text{-}space \to \beta :: topological\text{-}space) \to \mathbb{B}$
$continuous\text{-}on\ S\ f \Longleftrightarrow \forall x \in S.\ (f \longrightarrow f\ x)\ (at\ x\ within\ S)$

4.4 Compactness

An important topological concept is *compactness* of sets. There are different characterizations of compactness: sequential compactness, cover compactness and countable cover compactness. Unfortunately these characterizations are not equal on each topological space, but we will show in which type classes they are.

First we introduce *cover compactness*; it does not require any other topological concepts besides open sets. A *cover* of a set U is a set of open sets whose union is a superset of U. A set U is compact iff for each cover C there exists a finite subset of C which is also a cover:

$$\begin{aligned}&\mathit{compact} :: (\alpha :: \mathit{topological\text{-}space})\ \mathit{set} \to \mathbb{B}\\&\mathit{compact}\ U \longleftrightarrow\\&\quad (\forall C.\ (\forall c \in C.\ \mathit{open}\ c) \wedge U \subseteq \bigcup C \implies \exists D \subseteq C.\ \mathit{finite}\ D \wedge U \subseteq \bigcup D)\end{aligned}$$

Topology usually talks about compact spaces U, where the open sets are restricted to the topological space U, which would be $\mathcal{U}_\alpha$ in our case. This would not be very helpful, we would need to define a type for each compact space. Luckily, cover compactness works also with covers which are proper supersets, which will be the case when we use it.

Cover compactness can be expressed using filters. A space U is compact iff for each proper filter on U exists an $x \in U$, s.t. a neighborhood of x is contained in the filter.

lemma

$$\begin{aligned}&\mathit{compact}\ U \longleftrightarrow\\&\quad (\forall F > \bot.\ \mathit{eventually}\ (\lambda x.\ x \in U)\ F \implies (\exists x \in U.\ \mathit{nhds}\ x \sqcap F > \bot))\end{aligned}$$

Similarly to cover compactness we define *countably-compact*, where a set is compact iff for each *countable* cover exists a finite subcover. Then *compact* obviously implies *countably-compact*, the other direction holds at least for a *second-countable-topology* space.

With limits and filters, characterizations of compactness apart from cover or countable compactness are possible. One often used characterization of compactness is *sequential compactness*, where for each sequence on the compact space U, there exists a subsequence converging in U (a subsequence of X is defined by selecting increasing indices into X, *subseq* r states that r is strictly increasing).

$$\begin{aligned}&\mathit{seq\text{-}compact} :: \alpha\ \mathit{set} \to \mathbb{B}\\&\mathit{seq\text{-}compact}\ U \longleftrightarrow\\&\quad (\forall X.\ (\forall n.\ X\ n \in U) \implies \exists r.\ \mathit{subseq}\ r \wedge \exists x \in U.\ (X \circ r) \longrightarrow x)\end{aligned}$$

On a first countable topology sequential equals countable cover compactness. On a second countable topology sequential, countable cover, and cover compactness are equal.

lemma fixes $U :: (\alpha :: \mathit{first\text{-}countable\text{-}topology})\ \mathit{set}$
shows $\mathit{countably\text{-}compact}\ U \longleftrightarrow \mathit{seq\text{-}compact}\ U$

lemma fixes $U :: (\alpha :: \mathit{second\text{-}countable\text{-}topology})\ \mathit{set}$
shows $\mathit{compact}\ U \longleftrightarrow \mathit{seq\text{-}compact}\ U$
shows $\mathit{compact}\ U \longleftrightarrow \mathit{countably\text{-}compact}\ U$

5 Mathematical Analysis

Analysis works with infinite sequences and limits and develops concepts like differentiation and integration. As seen in the previous section, limits have been formalized generically for topological spaces. The formalization leading to differentiation and integration has largely been ported from Harrison's formalization in HOL Light [4] for the type $\mathbb{R}^n$. In this section, we present the generalization to our hierarchy of type classes. Following Fig. 1, we start with the type classes for metric spaces and then present the type classes for vector spaces, which culminate in Euclidean spaces.

5.1 Metric Spaces

Metric spaces are specializations of topological spaces: while topological spaces talk about *nearness*, metric spaces require to explicitly give a *distance* between elements. This distance then induces a notion of *nearness*: a set is open iff for every element in that set, one can give a distance within which every element is *near*, i.e. in the open set. The following type class formalizes open sets induced by a distance:

class *open-dist* = **fixes** *open* :: α *set* $\to \mathbb{B}$ **and** *dist* :: $\alpha \to \alpha \to \mathbb{R}$
  assumes *open* $U \longleftrightarrow (\forall x \in U.\ \exists e > 0.\ \forall y.\ \mathit{dist}\ x\ y < e \implies y \in U)$

If the distance is a metric, it induces a particular topological space, namely a metric space. It is a first countable space and satisfies the Hausdorff separation property, i.e. it is actually a T_2-space.

class *metric-space* = *open-dist* +
  assumes *dist* $x\ y = 0 \longleftrightarrow x = y$ **and** *dist* $x\ y \leq$ *dist* $x\ z +$ *dist* $y\ z$
instance *metric-space* $\subseteq$ *t2-space*, *first-countable-topology*

One aspect that makes real numbers an interesting metric space is the fact that they are *complete*, which means that every sequence where the elements get arbitrarily close converges. Such a sequence is called *Cauchy sequence*, and a metric space is complete iff every Cauchy sequence converges.

Cauchy :: $(\mathbb{N} \to \alpha ::$ *metric-space*$) \to \mathbb{B}$
Cauchy $X \longleftrightarrow (\forall e > 0.\ \exists M.\ \forall m, n \geq M.\ \mathit{dist}\ (X\ m)\ (X\ n) < e)$

complete :: $(\alpha ::$ *metric-space*$)$ *set* $\to \mathbb{B}$
complete $U \longleftrightarrow (\forall X.\ (\forall i.\ X\ i \in U) \wedge$ *Cauchy* $X \implies \exists x \in U.\ X \longrightarrow x)$

class *complete-space* = *metric-space* + **assumes** *complete* $\mathcal{U}_\alpha$

We have generalized Harrison's formalization of the Banach fixed point theorem to metric spaces and we completed a characterization of compactness on metric spaces with total boundedness: compact sets are the complete ones that can, for every $e > 0$, be covered by a finite number of balls with radius e.

lemma $\forall U ::$ ($\alpha ::$ *metric-space*) *set*. *compact* $U \longleftrightarrow$
  complete $U \wedge (\forall e > 0.\ \exists T.\ \mathit{finite}\ T \wedge U \subseteq \bigcup_{t \in T} \{s \mid \mathit{dist}\ s\ t < e\})$

One instance of complete metric spaces is the type of finite maps $\alpha \to_f (\beta :: \textit{complete-space})$: the distance of two finite maps f, g with domains F, G is given by $\max_{i \in F \cup G} (\textit{dist}\ (f\ i)\ (g\ i)) + (\textbf{if}\ F = G\ \textbf{then}\ 0\ \textbf{else}\ 1)$. Then every Cauchy sequence eventually stabilizes at one particular finite domain and then converges uniformly. Another example is the type of bounded continuous functions $(\alpha :: \textit{topological-space}) \to_{bc} (\beta :: \textit{complete-space})$. Equipped with the supremum distance, they form a complete metric space.

Heine-Borel Spaces. One can provide the convenient characterization that compact sets are exactly the bounded and closed sets on a metric space if the additional assumption that bounded sequences possess a convergent subsequence holds. We summarize this assumption in a type class, which allows for convenient access to the characterization and the theorems it implies. Euclidean spaces like $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{R}^n$ are examples of instances.

class *heine-borel* = *metric-space* +
  assumes *bounded* $(\bigcup_x \{X\ x\}) \implies \exists x, r.\ \textit{subseq}\ r \wedge (X \circ r) \longrightarrow x$
instance *heine-borel* $\subseteq$ *complete-space*

lemma $\forall U :: (\alpha :: \textit{heine-borel})\ \textit{set}.\ \textit{compact}\ U \iff \textit{bounded}\ U \wedge \textit{closed}\ U$

5.2 Vector Spaces

One aspect that is often abstracted away from products of real numbers is their property of being a vector space, i.e. a space where addition and scaling can be performed. Let us present in this section the definition of vector spaces, normed vector spaces, and how derivatives are generalized for normed vector spaces.

Definition. Usually, a vector space is defined on an Abelian group of vectors V, which can be scaled with elements of a field F, and where distributive and compatibility laws need to be satisfied by scaling and addition. The type class based approach restricts the number of type variables to one; we therefore use locales (Isabelle's module system for dealing with parametric theories [3]) to abstractly reason about vector spaces with arbitrary combinations of F and V (which may be of different types). We define the type class *real-vector* for the common usage of $\mathbb{R}$ for the field F: the type class *ab-group-add*, which formalizes an Abelian group, provides the operations for addition and additive inverse for the type of vectors α (subtraction is defined in terms of these operations).

class *real-vector* = *ab-group-add* + **fixes** $\cdot_R :: \mathbb{R} \to \alpha \to \alpha$
  assumes $r \cdot_R (a + b) = r \cdot_R a + r \cdot_R b$ **and** $(r + q) \cdot_R a = r \cdot_R a + q \cdot_R a$
    and $r \cdot_R (q \cdot_R a) = (r \cdot q) \cdot_R a$ **and** $1 \cdot_R a = a$

A generalization of the length of a vector of real numbers is given by the norm in a vector space. The norm induces a distance in a vector space. Similar to

open-dist, which describes how *dist* induces *open* sets, we describe here how the norm induces the distance.

class *dist-norm* = **fixes** *norm* :: $\alpha \to \mathbb{R}$ **and** $-$ :: $\alpha \to \alpha \to \alpha$
  assumes *dist* $x\ y$ = *norm* $(x - y)$

A *normed vector space* is then defined as a vector space *real-vector* together with the usual assumptions of a separating and positively scalable norm, for which the triangle equality holds. The distance for the instantiation as metric space and open sets for the topology are induced by *dist-norm* and *open-dist*, respectively. Then every normed vector space is a metric space.

class *real-normed-vector* = *real-vector* + *dist-norm* + *open-dist* +
  assumes *norm* $x = 0 \longleftrightarrow x = 0$ **and** *norm* $(r \cdot_R x)$ = $|r| \cdot$ *norm* x
    and *norm* $(x + y) \leq$ *norm* $x +$ *norm* y
instance *real-normed-vector* $\subseteq$ *metric-space*

We define a filter to describe that the norm tends to infinity (*at-infinity* = *filtermap norm at-top*). We have lemmas about limits of vector space operations – for example *LIM* x *in* F. $f\ x + g\ x$:> G for G = *nhds* L (if f and g converge) or G = *at-infinity* (if f or g tend to infinity) – and hence continuity.

Complete normed vector spaces are called Banach spaces; we provide an extra type class for them. For example bounded continuous functions (α :: *topological-space*) $\to_{bc}$ (β :: *real-normed-vector*) equipped with pointwise addition and scaling form a Banach space.

class *banach* = *complete-space* + *real-normed-vector*

Derivatives. The HOL Light formalization includes derivatives of functions from $\mathbb{R}^n$ to $\mathbb{R}^m$. This derivative is a linear mapping, it is called Fréchet derivative or total derivative, and its matrix is called the Jacobian matrix. Our type class based formalization allows us to generalize (in accordance with textbook mathematics) the notion of Fréchet derivative to arbitrary normed vector spaces *real-normed-vector*, where the derivative is a *bounded* linear approximation. The limit may be approached from within an arbitrary set s:

bounded-linear :: (α :: *real-normed-vector* $\to$ β :: *real-normed-vector*) $\to \mathbb{B}$
bounded-linear $f' \longleftrightarrow (f'\ (x + y) = f'\ x + f'\ y \wedge f'\ (a \cdot_R x) = a \cdot_R (f'\ x) \wedge$
  $(\exists K.\ \forall x.$ *norm* $(f'\ x) \leq K \cdot$ *norm* $x))$

FDERIV $\square\ \square : \square$:> $\square$ :: $(\alpha \to \beta) \to \alpha \to \alpha$ *set* $\to (\alpha \to \beta)$
FDERIV $f\ x : s$:> $f' \longleftrightarrow$ (*bounded-linear* $f' \wedge$
  $((\lambda y.$ *norm* $(f\ y - f\ x - f'\ (y - x))/$*norm* $(y - x)) \longrightarrow 0)$ (*at* x *within* s))

We have generalized Harrison's results about derivatives of arithmetic operations, and the chain rule for differentiation to *real-normed-vector* spaces.

We provide a set of rules *FDERIV-eq-intros* that allows to compute derivatives: each of the rules assumes composition of a differentiable function with an

additional function and matches a variable to the derivative, which has to be solved by Isabelle's rewrite engine. Consider e.g., the following rule where the first assumption has to be solved by a repeated application of *FDERIV-eq-intros* and the second assumption needs to be solved by the simplifier:

lemma
  assumes *FDERIV* $f\ x : s :> f'$ **and** $(\lambda x.\ r \cdot_R (f'\ x)) = D$
  shows *FDERIV* $(\lambda x.\ r \cdot_R (f\ x))\ x : s :> D$

Algebraic Vector Spaces. Further specializations of (normed) vector spaces are available by including multiplication of vectors for a *real-normed-algebra* or *real-normed-field*. The only instances currently used are real and complex numbers $\mathbb{R}$ and $\mathbb{C}$ so we will not go into more detail here.

5.3 Euclidean Spaces

Another abstraction with geometric intuition is given by an *inner product* on normed vector spaces: while the norm can be interpreted as the length of a vector, the inner product can be used to describe the angle between two vectors together with their lengths (the cosine of the angle is the inner product divided by the product of the lengths). *dist-norm* and *open-dist* specify the induced metric and topology. The inner product is used to induce a norm. An inner product is a commutative bilinear operation $\bullet$ on vectors, for which $0 \leq x \bullet x$ holds with equality iff $x = 0$.

class *real-inner* = *real-vector* + *dist-norm* + *open-dist* +
  fixes $\bullet :: \alpha \to \alpha \to \mathbb{R}$
  assumes *norm* $x = \sqrt{x \bullet x}$ **and** $x \bullet y = y \bullet x$
    and $(x + y) \bullet z = x \bullet z + y \bullet z$ **and** $(r \cdot_R x) \bullet y = r \cdot_R (x \bullet y)$
    and $0 \leq x \bullet x$ **and** $x \bullet x = 0 \iff x = 0$
instance *real-inner* $\subseteq$ *real-normed-vector*

For vector spaces with inner products, there is for example orthogonality of vectors formalized, i.e. vectors with inner product zero.

Finally, we introduce *Euclidean spaces* as spaces with inner product and a finite coordinate basis, that means a finite set of orthogonal vectors of length 1. In addition, the zero vector is characterized by zero "coordinates" with respect to the basis. Any Euclidean space is a Banach space with a perfect second countable topology and satisfies the Heine-Borel property:

class *euclidean-space* = *real-inner* +
  fixes *Basis* :: α *set*
  assumes *finite Basis* **and** *Basis* $\neq \emptyset$ **and** $(\forall u \in \mathit{Basis}.\ x \bullet u = 0) \iff x = 0$
    and $u \in \mathit{Basis} \implies v \in \mathit{Basis} \implies u \bullet v =$ **if** $u = v$ **then** 1 **else** 0
instance *euclidean-space* $\subseteq$ *perfect-space*, *second-countable-topology*, *banach*, *heine-borel*

Linear algebra has been ported from Harrison's basic formalization, which includes notions of independence and span of a set of vectors. We prove for example independence of the basis and that the basis spans the whole Euclidean space.

For functions between *euclidean-space*s, we have ported from HOL Light that the Fréchet derivative can be described as the Jacobian matrix, the mean value theorem, and Brouwer's fixed point theorem, which allows to prove that the derivative of an inverse function is the inverse of the derivative.

Moreover we have ported Harrison's formalization of the gauge (or Heinstock-Kurzweil) integral and related properties (linearity, monotone and dominated convergence, and the fundamental theorem of calculus).

Instances for the type class *euclidean-space* are real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$ and the Cartesian types $\mathbb{R}^\alpha$ where $\alpha :: \mathit{finite}$ (which are isomorphic to $\alpha \to \mathbb{R}$). One advantage of our type class based approach is that we can use the same formalizations of Euclidean space (e.g. of the integral) for the different types, whereas in HOL Light, one needs to project e.g. from $\mathbb{R}^1$ to $\mathbb{R}$.

5.4 Real Numbers

The type of real numbers $\mathbb{R}$ is a special instance of Euclidean spaces; some parts of our formalization are only available for this case. For a function on real numbers, one usually thinks of the "derivative" as the slope of the function (or of the linear approximation), we therefore use the constant *DERIV*:

$$\mathit{DERIV}\ \square\ \square :> \square \quad :: \quad (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$$
$$\mathit{DERIV}\ f\ x :> f' \iff \mathit{FDERIV}\ f\ x : \mathcal{U}_{\mathbb{R}} :> (\lambda x.\ f' \cdot x)$$

It turns out that the general formalization of limits with filters allows to conveniently express e.g. l'Hôpital's rules in Isabelle/HOL: if the denominator of a quotient tends to infinity, then the quotient tends to the quotient of the derivatives of nominator and denominator (if they exist).

lemma
  fixes $f\ g :: \mathbb{R} \to \mathbb{R}$
  assumes $\mathit{LIM}\ x\ \mathit{in}\ \mathit{at\text{-}top}.\ g\ x :> \mathit{at\text{-}top}$
    and $\mathit{eventually}\ (\lambda x.\ g'\ x \neq 0)\ \mathit{at\text{-}top}$
    and $\mathit{eventually}\ (\lambda x.\ \mathit{DERIV}\ f\ x :> f'\ x \wedge \mathit{DERIV}\ g\ x :> g'\ x)\ \mathit{at\text{-}top}$
    and $((\lambda x.\ f'\ x / g'\ x) \longrightarrow L)\ \mathit{at\text{-}top}$
  shows $((\lambda x.\ f\ x / g\ x) \longrightarrow L)\ \mathit{at\text{-}top}$

6 Summary

We used the type class mechanism in Isabelle/HOL to formalize a hierarchy of spaces often used in mathematical analysis: starting with topological spaces, over metric spaces to Euclidean spaces. As in mathematics, the intention of using a hierarchical structure is to share definitions and proofs.

The reuse occurs for the introduction of extended reals $\overline{\mathbb{R}}$, the spaces of bounded continuous functions $\alpha \to_{bc} \beta$, and finite maps $\alpha \to_f \beta$. The extended reals $\overline{\mathbb{R}}$ need to exploit the topological type classes, as they do not form a metric space. The bounded continuous function space $\alpha \to_{bc} \beta$ is a Banach space. Immler and Hölzl [7] apply them to the Banach fixed point theorem to prove the existence of unique solutions of ordinary differential equations. Immler [6] uses finite maps $\alpha \to_f \beta$ to construct stochastic processes via a projective limit.

Our approach still has the problem that all operations are defined on $\mathcal{U}_\alpha$. The usage of finite maps $\alpha \to_f \beta$ in [6] illustrates this. We need a metric space whose dimensionality depends on a variable inside of a proof. Luckily, the disjoint union of metric spaces can be extended to a metric space. But such a trick is not always applicable, i.e. this is not possible for normed vector spaces. This can only be avoided by adding a carrier set to each operation or by extending HOL.

Despite the last point, our work shows that Isabelle's type class system suffices to describe many abstract structures occurring in mathematical analysis.

Acknowledgements. We want to thank John Harrison and his colleagues for the development of HOL Light's multivariate analysis. Further we want to thank Amine Chaieb and Robert Himmelmann for porting it to Isabelle/HOL.

References

1. Fleuriot, J.D., Paulson, L.C.: Mechanizing nonstandard real analysis. LMS Journal of Computation and Mathematics 3, 140–190 (2000)
2. 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)
3. Haftmann, F., Wenzel, M.: Local theory specifications in Isabelle/Isar. In: Berardi, S., Damiani, F., de'Liguoro, U. (eds.) TYPES 2008. LNCS, vol. 5497, pp. 153–168. Springer, Heidelberg (2009)
4. Harrison, J.V.: A HOL theory of Euclidean space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
5. Hölzl, J., Heller, A.: Three chapters of measure theory in Isabelle/HOL. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 135–151. Springer, Heidelberg (2011)
6. Immler, F.: Generic construction of probability spaces for paths of stochastic processes in Isabelle/HOL. Master's thesis, TU München (October 2012)
7. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in Isabelle/HOL. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 377–392. Springer, Heidelberg (2012)
8. Joshi, K.D.: Introduction to General Topology. John Wiley and Sons (1983)
9. Lester, D.R.: Topology in PVS: continuous mathematics with applications. In: Second Workshop on Automated Formal Methods, AFM 2007, pp. 11–20 (2007)
10. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. MSCS, Interactive theorem proving and the form. of math. 21, 1–31 (2011)

Formal Reasoning about Classified Markov Chains in HOL

Liya Liu, Osman Hasan, Vincent Aravantinos, and Sofiène Tahar

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8, Canada
{liy_liu,o_hasan,vincent,tahar}@ece.concordia.ca

Abstract. Classified Markov chains have been extensively applied to model and analyze various stochastic systems in many engineering and scientific domains. Traditionally, the analysis of these systems has been conducted using computer simulations and, more recently, also probabilistic model-checking. However, these methods either cannot guarantee accurate analysis or are not scalable due to the unacceptable computation times. As an alternative approach, this paper proposes to reason about classified Markov chains using HOL theorem proving. We provide a formalization of classified discrete-time Markov chains with finite state space in higher-order logic and the formal verification of some of their widely used properties. To illustrate the usefulness of the proposed approach, we present the formal analysis of a generic LRU (least recently used) stack model.

1 Introduction

In analyzing the stationary behaviors of Markovian models, it is quite common to categorize Markov chains into different classes depending on the properties exhibited by their states [3]. Some commonly used classes include *reducible*, *irreducible*, *periodic*, *aperiodic*, *regular* and *absorbing Markov chains*. Classified Markov chains are very useful for the dynamic analysis of systems as their properties allow us to judge long-run characteristics of Markovain systems, such as if a system will re-visit a particular state or to determine the time of the first visit to a state. Some of the widely used application areas of the classified Markov chains are reliability analysis, performance analysis and validation of models.

Traditionally, simulation is the most commonly applied computer-based analysis technique for Markovian systems. The main idea here is to utilize an equilibrium vector to approximate $vp_{ij}^{(n)}$, where v is any probability vector and $p_{ij}^{(n)}$ is the n-step transition probability. The main reason behind using the equilibrium vector is the high computational costs associated with $vp_{ij}^{(n)}$ for large values of n. Moreover, many rounding errors also creep into the analysis due to the involvement of computer arithmetic. Such approximations and inaccuracies pose a serious problem while analyzing highly sensitive and safety-critical applications.

Due to the extensive usage of Markov chains for safety-critical systems, probabilistic model checking has been recently proposed for analyzing Markovian

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 295–310, 2013.

systems. Probabilistic model checking tools, such as *PRISM* [15], *VESTA* [16] and *Ymer* [19], provide precise system analysis by modeling the stochastic behaviors, including its random components, using probabilistic state machines and exhaustively verifying their probabilistic properties. However, some algorithms implemented in these model checking tools are based on numerical methods. For example, the Power method [14], which is a well-known iterative method, is applied to compute the steady-state probabilities (or limiting probabilities) of Markov chains in PRISM. For this reason, most of the stationary properties analyzed in model checkers are time bounded. Moreover, probabilistic model checking often utilizes unverified algorithms and optimization techniques. Finally, model checking cannot be used to verify generic mathematical expressions for probabilistic analysis. In order to overcome these limitations, we proposed to use higher-order-logic theorem proving for analyzing Discrete Time Markov Chains (DTMCs) [10], where we presented a formal definition of DTMC and verified some of its properties using the HOL theorem prover. This formalization enabled us to formally analyze some simple Markovian models in HOL. In order to extend the capabilities of higher-order-logic theorem proving based analysis of Markovian models and thus be able to analyze a wider range of real-world systems, we present in the current paper the formalization of classified DTMCs.

In [8], the authors formally defined a time-homogeneous Markov chain based on the state space and the transition matrix in Isabelle/HOL, and they assumed no initial distribution or start state. Compared to their definition and the formalization presented in [10], which was based on the probability theory developed in [6], this paper describes a more generic higher-order-logic formalization of finite-state DTMC. Our work is based on a more general formalization of probability theory [11], which provides us with the flexibility to model inhomogeneous DTMCs or several random processes (involving DTMCs) containing distinct types of state spaces. We then build upon the formal DTMCs to formalize classified DTMCs and to formally verify the properties of aperiodic and irreducible DTMCs. For illustration purposes, we formally validate a least recently used (LRU) stack model using our formalization.

2 Formalization of DTMCs

A *probability space* is a measure space $(\Omega, \Sigma, \mathcal{P}r)$ such that $\mathcal{P}r(\Omega) = 1$ [3]. Σ is a collection of subsets of Ω (these should satisfy some closure axioms that we do not specify here) which are called *measurable sets*. In [12], a higher-order logic probability theory is developed, where given a probability space `p`, the functions `space` and `subsets` return the corresponding Ω and Σ, respectively. Mathematically, a *random variable* is a measurable function between a probability space and a *measurable space*, which refers to a pair $(S, \mathcal{A})$, where S is a set and $\mathcal{A}$ is a σ-algebra, i.e., a collection of subsets of S satisfying some particular properties [3]. In HOL, we write `random_variable X p s` to state that a function `X` is a random variable on a probability space `p` and the measurable outcome space `s`. Building on these foundations, measure theoretic formalizations of probability, Lebesgue integral and information theories are presented in [12].

A *stochastic process* [3] is a function $X : T \to \Omega$ where $T = \mathbb{N}$ (*discrete-time process*) or $T = \mathbb{R}$ (*continuous-time process*) and Ω is a measurable set called the *state space* of X. A *(finite-state) DTMC* is a discrete-time stochastic process that has a finite Ω and satisfies the *Markov property* [4]: for $0 \leq t_0 \leq \cdots \leq t_n$ and $f_0, \cdots, f_{n+1}$ in the state space, then: $\mathcal{P}r\{X_{t_{n+1}} = f_{n+1} | X_{t_n} = f_n, \ldots, X_{t_0} = f_0\} = \mathcal{P}r\{X_{t_{n+1}} = f_{n+1} | X_{t_n} = f_n\}$.

This allows to formalize the Markov property as follows:

Definition 1 (Markov Property).

```
⊢ ∀ X p s. mc_property X p s =
   (∀ t. random_variable (X t) p s) ∧
   ∀ f t n.
    increasing_seq t ∧ ℙ(⋂ₖ∈ [0,n−1]{x | X tₖ x = f k}) ≠ 0 ⇒
    (ℙ({x | X tₙ₊₁ x = f (n + 1)}|{x | X tₙ x = f n} ∩
                                   ⋂ₖ∈ [0,n−1]{x | X tₖ x = f k}) =
     ℙ({x | X tₙ₊₁ x = f (n + 1)}|{x | X tₙ x = f n}))
```

where `increasing_seq t` is defined as `∀ i j. i < j ⇒ t i < t j`, thus formalizing the notion of increasing sequence. The first conjunct indicates that the Markov property is based on a random process $\{X_t : \Omega \to S\}$. The quantified variable `X` represents a function of the random variables associated with time `t` which has the type `num`. This ensures the process is a *discrete time* random process. The random variables in this process are the functions built on the probability space `p` and a measurable space `s`. The conjunct $\mathbb{P}(\bigcap_{\mathtt{k}\in[0,\mathtt{n}-1]}\{\mathtt{x} \mid \mathtt{X}\ \mathtt{t_k}\ \mathtt{x} = \mathtt{f}\ \mathtt{k}\}) \neq 0$ ensures that the corresponding conditional probabilities are well-defined, where `f k` returns the $\mathtt{k}^{th}$ element of the state sequence.

A DTMC is usually expressed by specifying: an initial distribution p_0 which gives the probability of initial occurrence $\mathcal{P}r(X_0 = s) = p_0(s)$ for every state; and transition probabilities $p_{ij}(t)$ which give the probability of going from i to j for every pair of states i, j in the state space [13]. For states i, j and a time t, the *transition probability* $p_{ij}(t)$ is defined as $\mathcal{P}r\{X_{t+1} = j | X_t = i\}$, which can be easily generalized to *n-step transition probability*.

$$p_{ij}^{(n)} = \begin{cases} \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} & n = 0 \\ \mathcal{P}r\{X_{t+n} = j | X_t = i\} & n > 0 \end{cases}$$

This is formalized in HOL as follows Definition 2 below so that the discrete-time Markov chain (DTMC) can be formalized as Definition 3:

Definition 2 (Transition Probability).

```
⊢ ∀ X p s t n i j.
  Trans X p s t n i j =
  if n = 0 then
    if i ∈ space s ∧ j ∈ space s then
      if (i = j) then 1 else 0
```

```
    else 0
  else ℙ({x | X (t + n) x = j}|{x | X t x = i})
```

Definition 3 (DTMC).

```
⊢ ∀ X p s Linit Ltrans. dtmc X p s Linit Ltrans =
  mc_property X p s ∧ (∀ i. i ∈ space s ⇒ {i} ∈ subsets s) ∧
  ∀ i. i ∈ space s ⇒ (Linit i = ℙ{x | X t x = i}) ∧
  ∀ t i j. (ℙ{x | X t x = i} ≠ 0) ⇒
           (Ltrans t i j = Trans X p s t 1 i j)
```

It is important to note that `X` is polymorphic, i.e., it is not constrained to a particular type, which is a very useful feature of our definition.

In practice, many applications actually make use of *time-homogenous DTMCs*, i.e., DTMCs with finite state-space and time-independent transition probabilities [2]. They can be formalized as follows:

Definition 4 (Time homogeneous DTMC).

```
⊢ ∀ X p s p₀ pᵢⱼ. th_dtmc X p s p₀ pᵢⱼ =
   dtmc X p s p₀ pᵢⱼ ∧ FINITE (space s) ∧
   ∀ t i j. Trans X p s (t + 1) 1 i j = Trans X p s t 1 i j
```

For time-homogenous DTMCs, $\forall\ t\ t'.\ p_{ij}(t) = p_{ij}(t')$ and thus $p_{ij}(t)$ is simply written as p_{ij}.

It is often the case that we are interested in the probability of some specific states as time tends to infinity under certain conditions. This is the main reason why stationary behaviors of stochastic processes are frequently analyzed in engineering and scientific domains. There is no exception for Markovian systems.

Let $\{X_t\}_{t\geq 0}$ be a Markov chain having state space Ω and transition probability p_{ij} for going from a state with value i to a state with value j. If $\pi(i)$, $i \in \Omega$, are nonnegative numbers summing to one, and if $j \in \Omega$, then $\pi(j) = \sum_{i\in\Omega} \pi(i)p_{ij}$ is called a *stationary distribution*. The corresponding HOL definition is as follows.

Definition 5 (Stationary Distribution).

```
⊢ ∀ p X f s. stationary_dist p X f s =
  ∑ (f k) = 1 ∧
  ∀ i. i ∈ space s ⇒
       0 < f i ∧ ∀ t. f i = ∑ f k * Trans X p s t 1 k i
```

Using these fundamental definitions, we formally verified most of the classical properties of DTMCs with finite state-space in HOL. Some of the relevant ones to the context of this paper are presented later.

3 Formalization of Classified DTMCs

In this section, we first formalize some foundational notions of classified Markov chains. Then, we use these results along with our formal definition of DTMC to formalize classified Markov chains. The foremost concept of states classification is the *first passage time* τ_j, or the *first hitting time*, which is defined as the

minimum time required to reach a state j from the initial state i:

$$\tau_j = min\{t > 0 : X_t = j\}.$$

The first passage time can be defined in HOL as:

Definition 6 (First Passage Time).

```
⊢ FPT X x j = MINSET {t | 0 < t ∧ (X t x = j)}
```

where `X` is a random process and `x` is a sample in the probability space associated with the random variable X_t. Note that the first passage time is also a random variable.

The conditional distribution of τ_j defined as the probability of the events starting from state i and visiting state j at time n is expressed as $f_{ij}^{(n)} = \mathcal{P}r\{\tau_j = n|X_0 = i\}$. It can be formalized in HOL as follows:

Definition 7 (Probability of First Passage Events).

```
⊢ f X p i j n = ℙ({x | FPT X x j = n}|{x | X 0 x = i})
```

Another important notion, denoted as f_{ij}, is the probability of the events starting from state i and visiting state j at all times n, is expressed as $f_{ij} = \sum_{n=1}^{\infty} f_{ij}^{(n)}$. It can be expressed in HOL as `(λ n. f X p i j n) sums` f_{ij}. Another interesting concept is f_{jj}, which provides the probability of events starting from state j and eventually returning back to j. If $f_{jj} = 1$, then the *mean return time* of state j is defined as $\mu_j = \sum_{n=1}^{\infty} n f_{jj}^{(n)}$. The existence of this infinite summation can be specified as `summmable (λ n. n * f X p j j n)` in HOL.

A state j in a DTMC $\{X_t\}_{t\geq 0}$ is called *transient* if $f_{jj} < 1$, and *persistent* if $f_{jj} = 1$. If the mean return time μ_j of a persistent state j is finite, then j is said to be *persistent nonnull state* (or *positive persistent state*). Similarly, if μ_j is infinite, then j is termed as *persistent null state*.

The greatest common divisor (*gcd*) of a set is a frequently used mathematical concept in defining classified states. We formalize the gcd of a set as follows:

Definition 8 (gcd of a Set).

```
⊢ GCD_SET A = MAXSET {r | ∀ x. x ∈ A ⇒ divides r x}
```

For a state j, a *period* of j is any n such that $p_{jj}^{(n)}$ is greater than 0. We write $d_j = gcd\ \{n : p_{jj}^{(n)} > 0\}$ as the gcd of the set of all periods.

A state i is said to be *accessible* from a state j (written $j \rightarrow i$), if the n-step transition probability of the events from state i to j is nonzero. Two states i, j are called *communicating states* (written $i \leftrightarrow j$) if they are mutually accessible. A state j is an *absorbing state* if $p_{jj} = 1$. The formalization of some other foundational notions of classified states is given in Table 1. Now, we build upon the above mentioned definitions to formalize classified DTMCs. Usually, a DTMC is said to be *irreducible* if every state in its state space can be reached from any other state including itself in finite steps.

Table 1. Formalization of Classified States

Definition	Condition	HOL Formalization
Transient State	$f_{jj} < 1$	⊢ ∀ X p j. Transient_state X p j = ∀ x. {t | 0 < t ∧ (X t x = j)} ≠ ∅ ∧ (∃ s. s < 1 ∧ (λ n. f X p j j n) sums s)
Persistent State	$f_{jj} = 1$	⊢ ∀ X p j. Persistent_state X p j = ∀ x. {t | 0 < t ∧ (X t x = j)} ≠ ∅ ∧ (λ n. f X p j j n) sums 1
Persistent Nonnull State	$f_{jj} = 1$ $\mu_j < \infty$	⊢ ∀ X p j. Nonnull_state X p j = Persistent_state X p j ∧ summable (λ n. n * f X p j j n)
Persistent Null State	$f_{jj} = 1$ $\mu_j = \infty$	⊢ ∀ X p j. Null_state X p j = Persistent_state X p j ∧ ∼ summable (λ n. n * f X p j j n)
Periods of a State	$0 < \mathrm{n}$ $0 < p^n_{jj}$	⊢ ∀ X p s j. Period_set X p s j = {n | 0 < n ∧ ∀ t. 0 < Trans X p s t n j j}
GCD of a Period Set	d_j	⊢ ∀ X p s j. Period X p s j = GCD_SET (Period_set X p s j)
Periodic State	$d_j > 1$	⊢ ∀ X p s j. Periodic_state X p s j = (1 < Period X p s j) ∧ (Period_set X p s j ≠ ∅)
Aperiodic State	$d_j = 1$	⊢ ∀ X p s j. Aperiodic_state X p s j = (Period X p s j = 1) ∧ (Period_set X p s j ≠ ∅)
Accessibility	$\mathrm{i} \to \mathrm{j}$	⊢ ∀ X p s i j.Accessibility X p s i j = ∀ t. ∃ n. 0 < n ∧ 0 < Trans X p s t n i j
Communicating State	$\mathrm{i} \leftrightarrow \mathrm{j}$	⊢ ∀ X p s i. Communicating_states X p s i j = (Accessibility X p s i j) ∧ (Accessibility X p s j i)
Absorbing State	$p_{jj} = 1$	⊢ ∀ X p s j. Absorbing_states X p s j = (Trans X p s t 1 j j = 1)

Definition 9 (Irreducible DTMC).

```
⊢ Irreducible_mc X p s p0 pij =
   th_dtmc X p s p0 pij ∧
  (∀ i j. i ∈ space s ∧ j ∈ space s ⇒
          Communicating_states X p s i j)
```

whereas if there exists a state in the state space of a DTMC, which cannot reach some other states, then this DTMC is called *reducible*.

Definition 10 (Reducible DTMC).

```
⊢ Reducible_mc X p s p0 pij =
  th_dtmc X p s p0 pij ∧
  (∃ i j. i ∈ space s ∧ j ∈ space s ∧
         ∼ Communicating_states X p s i j)
```

A DTMC is considered as *aperiodic* if every state in its state space is an aperiodic state; otherwise it is a *periodic DTMC*.

Definition 11 (Aperiodic DTMC).

```
⊢ Aperiodic_mc X p s p0 pij =
  th_dtmc X p s p0 pij ∧
  ∀ i. i ∈ space s ⇒ Aperiodic_state X p s i
```

Definition 12 (Periodic DTMC).

```
⊢ Periodic_mc X p s p0 pij =
  th_dtmc X p s p0 pij ∧
  ∃ i. i ∈ space s ∧ Periodic_state X p s i
```

If at least one absorbing state exists in a DTMC and it is possible to go to the absorbing state from every non-absorbing state, then such a DTMC is named as *absorbing DTMC*.

Definition 13 (Absorbing DTMC).

```
⊢ Absorbing_mc X p s p0 pij =
  th_dtmc X p s p0 pij ∧
  ∃ i. i ∈ space s ∧ Absorbing_state X p s i ∧
       ∀ j. j ∈ space s ⇒ Communicating_state X p s i j
```

Finally, if there exists some n such that $p_{ij}^{(n)} > 0$ for all states i and j in a DTMC, then this DTMC is defined as a *regular DTMC*.

Definition 14 (Regular DTMC).

```
⊢ Regular_mc X p s p0 pij =
  th_dtmc X p s p0 pij ∧
  ∃ n. ∀ i j. i ∈ space s ∧ j ∈ space s ⇒
               Trans X p s t n i j > 0
```

To the best of our knowledge, the above mentioned definitions constitute the first formalization of classified DTMCs in higher-order logic. Their main utility is to formally specify and analyze the dynamic features of Markovian systems within a sound theorem prover as will be demonstrated in Section 5.

4 Verification of DTMC Properties

In this section, we utilize the definitions given above, to verify some of the most frequently used properties of DTMCs and classified DTMCs. The formal verification of these properties not only ensure the correctness of our definitions but also plays a vital role in formal reasoning about DTMCs and classified DTMCs in a theorem prover.

4.1 DTMC Properties

The *joint probability distribution* of a DTMC is the probability of a chain of states to occur. It is very useful in analyzing multi-stage experiments. In addition, this concept is the basis for the frequently used joint probability generating functions.

Theorem 1 (Joint Probability Distribution).

A joint probability distribution of n discrete random variables $X_0, \ldots X_n$ *in a finite DTMC* $\{X_t\}_{t\geq 0}$ *satisfies:*

$$\mathcal{P}r(X_t = L_0, \cdots, X_{t+n} = L_n) = \prod_{k=0}^{n-1} \mathcal{P}r(X_{t+k+1} = L_{k+1} | X_{t+k} = L_k) \mathcal{P}r(X_t = L_0)$$

```
⊢ ∀ X p s p L p₀ pᵢⱼ n.
   dtmc X p s p₀ pᵢⱼ ⇒
   ℙ(⋂_{k=0}^{n} {x | X (t + k) x = EL k L}) =
   ∏_{k=0}^{n-1} ℙ({x | X (t + k + 1) x = EL (k + 1) L} |
          {x | X (t + k) x = EL k L}) ℙ{x | X t x = EL 0 L}
```

The proof of Theorem 1 is based on induction on the variable n, Definition 3 and some arithmetic reasoning.

The Chapman-Kolmogorov equation [3] is a widely used property of time homogeneous DTMCs. It basically gives the probability of going from state i to j in $m + n$ steps. Assuming the first m steps take the system from state i to some intermediate state k and the remaining n steps then take the system from state k to j, we can obtain the desired probability by adding the probabilities associated with all the intermediate steps.

Theorem 2 (Chapman-Kolmogorov Equation).

For a finite time homogeneous DTMC $\{X_t\}_{t\geq 0}$*, its transition probabilities satisfy the Chapman-Kolmogorov Equation*

$$p_{ij}^{(m+n)} = \sum_{k\in\Omega} p_{ik}^{(m)} p_{kj}^{(n)}$$

```
⊢ ∀ X p s i j t m n p₀ pᵢⱼ.
   th_dtmc X p s p₀ pᵢⱼ ⇒
   Trans X p s t (m + n) i j =
   ∑_{k∈space s} (Trans X p s t m i k * Trans X p s t n k j)
```

The proof of Theorem 2 again involves induction on the variables m and n and both of the base and step cases are discharged using the following lemma:

Lemma 1 (Multistep Transition Probability).

```
⊢ ∀ X p s i j t m p₀ pᵢⱼ.
   th_dtmc X p s p₀ pᵢⱼ ⇒
   Trans X p s t (m + 1) i j =
   ∑_{k∈space s} (Trans X p s t 1 k j * Trans X p s t m i k)
```

The proof of Lemma 1 is primarily based on Definitions 3 and 4 and the additivity property of probabilities.

The unconditional probabilities associated with a Markov chain are called *absolute probabilities*, which can be computed by applying the initial distributions and n-step transition probabilities. From now, let us write $p_i^{(n)}$ for the probability $\mathcal{P}r(X_n = j)$. We then have the following result:

Theorem 3 (Absolute Probability).

In a finite time homogeneous DTMC, the absolute probabilities $p_j^{(n)}$ *satisfy*

$$p_j^{(n)} = \mathcal{P}r(X_n = j) = \sum_{k \in \Omega} \mathcal{P}r(X_0 = k)\mathcal{P}r(X_n = j | X_0 = k)$$

```
⊢ ∀ X p s j n p₀ pᵢⱼ.
   th_dtmc X p s p₀ pᵢⱼ ⇒
   ℙ{x | X n x = j} =
   ∑_{k∈space s} ℙ{x | X 0 x = k}ℙ({x | X n x = j}|{x | X 0 s = k})
```

The proof of Theorem 3 is based on the Total Probability theorem along with some basic arithmetic and probability theoretic reasoning.

The formal proof script for the above mentioned properties and many other useful properties is available at [9].

4.2 Classified DTMC Properties

Among the classified DTMCs formalized in the previous section, `aperiodic and irreducible DTMCs` are considered to be the most widely used ones in analyzing Markovian systems because of their attractive stationary properties, i.e., their limit probability distributions are independent of the initial distributions. For this reason, we now focus on the verification of some key properties of aperiodic and irreducible DTMCs [5].

Theorem 4 (Closed Period Set).

In an aperiodic DTMC, the set of the times when state i has a non-null probability of being visited is closed under addition.

```
⊢ ∀ X p s p₀ pᵢⱼ i.
   Aperiodic_DTMC X p s p₀ pᵢⱼ ∧ i ∈ space s ⇒
   ∀ a b. a ∈ Period_set X p s i ∧ b ∈ Period_set X p s i ⇒
          (a + b) ∈ Period_set X p s i
```

We verified the above theorem by using Theorem 2 and arithmetic and set theoretic reasoning.

Another key property of an aperiodic DTMC states that the transition probability $p_{ij}^{(n)}$ is greater than zero, for all states i and j in its state space, after n steps. It is very useful in analyzing the stability or reliability of real-world systems.

Theorem 5 (Positive Return Probability).

For any state i in the finite state space S of an aperiodic DTMC, there exists an $N < \infty$ *such that* $0 < p_{ii}^{(n)}$, *for all* $n \geq N$.

```
⊢ ∀ X p s p₀ pᵢᵢ i t.
   Aperiodic_DTMC X p s p₀ pᵢᵢ ∧ i ∈ space s ⇒
   ∃ N. ∀ n. N ≤ n ⇒ 0 < Trans X p s t n i i
```

The formal reasoning about the correctness of the above theorems involves Theorems 2 and 4 and the following lemmas, along with some arithmetic reasoning and set theoretic reasoning.

Lemma 2 (Positive Element in a Closed Set).

If an integer set S contains at least one nonzero element and S is closed under addition and subtraction, then $S = \{kc; k \in \mathbb{Z}\}$, where c is the least positive element of S.

```
⊢ ∀ s:int → bool. s ≠ ∅ ∧
   (∀ a b. a ∈ s ∧ b ∈ s ⇒ (a + b) ∈ s ∧ (a - b) ∈ s) ⇒
   0 < MINSET {r | 0 < r ∧ r ∈ s} ∧
   (s = {r | ?k. r = k * MINSET {r | 0 < r ∧ r ∈ s}})
```

Lemma 3 (Linearity of Two Integer Sequences).

For a positive integer sequence $a_1, a_2, \cdots, a_k$, there exists an integer sequence $n_1, n_2, \cdots, n_k$, such that $d = \sum_{i=1}^{k} n_i a_i$, where d is the greatest common divisor of sequence $a_1, a_2, \cdots, a_k$.

```
⊢ ∀ a k. 0 < k ∧ (∀ i. i ≤ k ⇒ 0 < a i) ⇒
   (∃ n. GCD_SET {a i | i ∈ [0, k]} = ∑ᵢ₌₀ᵏ n i * a i)
```

Lemma 4 (Least Number).

If a set of positive integers A is nonlattice, i.e., its gcd is 1, and closed under addition, then there exists an integer $N < \infty$ such that $n \in A$ for all $N \leq n$.

```
⊢ ∀ (A:int → bool) a.
   (A = {a i | 0 < a i ∧ i ∈ UNIV(:num)}) ∧ (GCD_SET A = 1) ∧
   (∀a b. a ∈ A ∧ b ∈ A ⇒ (a + b) ∈ s) ⇒ (∃N. {n | N ≤ n} ⊂ A)
```

The proofs of Lemmas 2, 3 and 4 are based upon various summation properties of integer sets. These properties are not available in the HOL libraries and thus had to be verified as part of our development.

Theorem 6 (Existence of Positive Transition Probabilities).

For any aperiodic and irreducible DTMC with finite state space S, there exists an N, for all $n \geq N$, such that the n-step transition probability $p_{ij}^{(n)}$ is non-zero, for all states i and $j \in S$.

```
⊢ ∀ X p s p₀ pᵢⱼ i j t.
   Aperiodic_DTMC X p s p₀ pᵢⱼ ∧ Irreducible_DTMC X p s p₀ pᵢⱼ ∧
   i ∈ space s ∧ j ∈ space s ⇒
   ∃ N. ∀ n. N ≤ n ⇒ 0 < Trans X p s t n i j
```

We proceed with the proof of Theorem 6 by performing case analysis on the equality of i and j. The rest of the proof is primarily based on Theorems 2 and 5, Definition 1 and Lemmas 3 and 4.

Theorem 7 (Existence of Long-run Transition Probabilities).

For any aperiodic and irreducible DTMC with finite state space S and transition probabilities p_{ij}, there exists $\lim_{n\to\infty} p_{ij}^{(n)}$, for all states i and $j \in S$.

```
⊢ ∀ X p s p₀ pᵢⱼ i j t.
   Aperiodic_DTMC X p s p₀ pᵢⱼ ∧ Irreducible_DTMC X p s p₀ pᵢⱼ ⇒
   ∃ u. (λ n. Trans X p s t n i j → u)
```

We firstly prove the monotonic properties of M_j^n and m_j^n, which are the maximum and minimum values of the set $\{n \leq 1: p_{ij}^{(n)}>0\}$, respectively. Then, the proof is completed by verifying the convergence of the sequence $(M_j^n$ - $m_j^n)$ for all n by applying Theorem 2 and some properties of real sequences. It is important to note that we do not need to use the assumption $j \in$ *space* s here, like all the other theorems, as $\forall\ n\ j.\ j \notin$ *space* $s \Rightarrow (p_j^{(n)} = 0)$, which in turn implies $\lim_{n\to\infty} p_j^{(n)} = 0$ and $\lim_{n\to\infty} p_{ij}^{(n)} = 0$. The long-run probability distributions are often considered in the convergence analysis of random variables in stochastic systems. It is not very easy to verify that the limit probability distribution of a certain state exists in a generic non-trivial DTMC, because the computations required in such an analysis are often tremendous. However, in the aperiodic and irreducible DTMCs, we can prove that all states possess limiting probability distribution, by the following two theorems.

Theorem 8 (Existence of Long-run Probability Distributions).

For any aperiodic and irreducible DTMC with finite state space S, there exists $\lim_{n\to\infty} p_i^{(n)}$, for any state $i \in S$.

```
⊢ ∀ X p s p₀ pᵢⱼ i.
   Aperiodic_DTMC X p s p₀ pᵢⱼ ∧ Irreducible_DTMC X p s p₀ pᵢⱼ ⇒
   ∃ u. (λ n. ℙ{x | X n x = i} → u)
```

We used Theorems 3 and 7, along with some properties of the limit of a sequence, to prove this theorem in HOL.

Theorem 9 (Existence of Steady State Probability).

For every state i in an aperiodic and irreducible DTMC, $\lim_{n\to\infty} p_i^{(n)}$ is a stationary distribution.

```
⊢ ∀ X p s p₀ pᵢⱼ.
   Aperiodic_DTMC X p s p₀ pᵢⱼ ∧ Irreducible_DTMC X p s p₀ pᵢⱼ ⇒
    (stationary_dist p X (λ i. lim ℙ{x | X n x = i}) s)
                              n→∞
```

The proof of Theorem 9 involves rewriting with Definition 5 and then splitting it into the following three subgoals:

- $0 \leq \lim_{n\to\infty} p_j^{(n)}$
- $\sum_{i\in\Omega} \lim_{n\to\infty} p_i^{(n)} = 1$
- $\lim_{n\to\infty} p_j^{(n)} = \sum_{i\in\Omega} \lim_{n\to\infty} p_i^{(n)} p_{ij}$

Utilizing the probability bounds theorem, we can prove the first subgoal. The proof of the second subgoal is primarily based on the additivity property of conditional probability [7]. Then the last subgoal can be proved by applying the linearity of limit of a sequence and the linearity of real summation.

All theorems presented in this section would facilitate the formal reasoning about the system properties that can be modeled using classified Markov chains. For illustration purposes, we present the formal analysis of a LRU stack model in the next section.

5 Formal Validation of LRU Stack Model

With the rapid development of computer technology, cache memory management becomes indispensable in computer architectures. The memory reference behaviors of various programs is one of the main deciding factors in designing efficient virtual memory operating systems. The Least Recently Used (LRU) stack model describes a behavior of reference strings where the probability of referencing a given page i at time t depends on the pages referenced in the closest past. In [1], the authors assumed the distance string for referencing a page as a sequence of independent identically distributed (IID) random variables in their LRU stack model, which was described as an aperiodic and irreducible DTMC [18]. However, in [17], the authors argued that the model constructed in [1] was not able to correctly depict the behavior of the LRU algorithm in multiprogramming. We want to formally verify the results of the latter authors using our formalization of aperiodic and irreducible DTMC.

5.1 LRU Stack Model

In a Least Recently Used (LRU) stack model, as shown in Figure 1, a sequence of stacks $s_1 s_2 \cdots s_t \cdots$ are associated with a reference string $w = \mathrm{x}_1\mathrm{x}_2\cdots\mathrm{x}_{t+1}\cdots$. Any stack s_t is a n-tuple $(j_1, j_2, \cdots, j_n)$, where j_i refers to the ith most recently referenced page at time t [18]. Let D_t be the position of the page x_t in the stack s_{t-1}. Then the distance string is $\mathrm{D}_1\mathrm{D}_2\cdots\mathrm{D}_t\cdots$, which is associated with the referencing string. This distance string can be modeled as a sequence of independent and identically distributed (IID) random variables [1], which makes their probability mass function (PMF) as $\mathcal{P}r(D_t = i) = \mathrm{a}_i$, where $i = 1, 2, \ldots,$ n and refers to the position of the least recently used page in the stack at time t, and $\sum_{j=1}^{n} \mathrm{a}_j = 1$. This way the distribution function becomes $\mathcal{P}r(D_t \leq i) = \sum_{j=1}^{i} \mathrm{a}_j$. If a tagged page occupies the ith position in the stack at time t, which

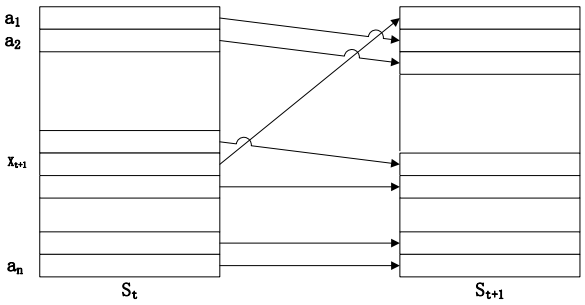


Fig. 1. LRU Stack updating Procedure [18]

is expressed as s_t in Figure 1, then the position of this page in the stack s_{t+1} depends on the next reference x_{t+1} and the position of this page in the stack s_t.

Based on the described updating procedure in the LRU stack, the evaluation of the page-fault rate of the LRU paging algorithm becomes quite simple. If the evaluated program has been allocated i page frames of main memory, then a page fault will occur at time t when $D_t > i$. Hence, the page fault probability is

$$\mathcal{F}(LRU) = \mathcal{P}r(D_t > i) = 1 - \sum_{j=1}^{i} a_j$$

The movement of the tagged page through the LRU state is then a random process $\{E_t\}_{t\geq 0}$. If the page occupies the ith position in stack s_t, then $E_t = i$, for all i, $1 \leq i \leq n$. Now, we have the following transition probabilities [18]:

$$p_{i1} = \mathcal{P}r(E_{t+1} = 1 | E_t = i) = \mathcal{P}r(D_{t+1} = i) = a_i, 1 \leq i \leq n$$

$$p_{ii} = \mathcal{P}r(E_{t+1} = i | E_t = i) = \mathcal{P}r(D_{t+1} < i) = \sum_{j=1}^{i-1} a_{j-1}, 2 \leq i \leq n$$

$$p_{i,i+1} = \mathcal{P}r(E_{t+1} = i+1 | E_t = i) = \mathcal{P}r(D_{t+1} > i) = 1 - \sum_{j=1}^{i} a_{j-1}, 1 \leq i \leq n-1$$

$$p_{i,j} = 0, otherwise.$$

The LRU stack is then described as an aperiodic and irreducible DTMC by assuming $a_i > 0$ for all $i \in [1, n]$ [18]. The state diagram of this aperiodic and irreducible DTMC is shown in Figure 2, where we can find that the transition probabilities can be expressed as the following higher-order logic function [18]:

Definition 15 (Transition Probability Matrix).

```
⊢ Lt a t i j =
  if (j = 1) then a i else
  if (j - i = 1) then 1 - ∑ i j=1 a j else
  if (j = i) then ∑ i−1 j=1 a j else 0
```

which can be used to formalize the LRU stack model as:

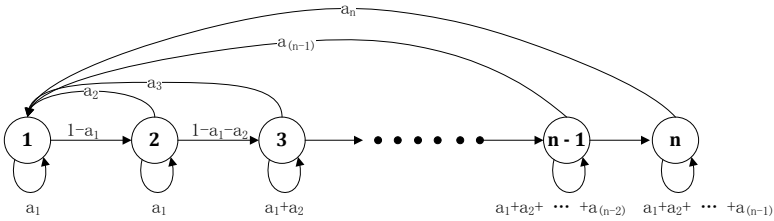


Fig. 2. The State Diagram for the LRU Stack Model

Definition 16 (LRU Model).

```
⊢ LRU_model X p a n p0.
  Aperiodic_DTMC X p ([1, n], POW ([1, n])) p0 (Lt a) ∧
  Irreducible_DTMC X p ([1, n], POW ([1, n])) p0 (Lt a) ∧
  1 <= n ∧ (∀ j. 0 < j ∧ j ≤ n ⇒ 0 < a j) ∧ (∑_{j=1}^{n} a j = 1)
```

where the state space is described as a pair `([1, n], POW ([1, n]))`, in which the first element contains all the states $\{1, 2, \cdots, n\}$ and the second one is the sigma algebra of the first element. The condition $(1 \leq n)$ is used to avoid the case when the length of the referencing string is zero. The other two conditions represent the specification of the model mentioned above.

5.2 Verification of the Property

Using the formal definition of this LRU stack model, we can now formally reason about its limiting distributions, which are mainly used to describe the stationary behaviors of this model.

Theorem 10 (Existence of LRU in the Limiting State Distribution).

In the LRU stack model, there exists $\lim_{t\to\infty} p_i^{(n)}$*, for every i ∈ [1, n].*

```
⊢ ∀ X p a n p0 i.
  LRU_model X p a n p0 ∧ i ∈ [1, n] ⇒
  ∃ u. (λ t. ℙ{x | X t x = i} → u)
```

We verified this property by directly applying Theorem 8 and the definition of limit of a real sequence.

Theorem 11 (LRU Stationary Limiting State Distribution).

In the LRU stack model, $\lim_{t\to\infty} p_i^{(n)} = \frac{1}{n}$*, for every i ∈ [1, n].*

```
⊢ ∀ X p a n p0 i.
  LRU_model X p a n p0 ∧ i ∈ [1,n] ⇒ lim_{t→∞} ℙ{x | X t x = i} = 1/n
```

The proof of this property is primarily based on Theorems 3 and 11 along with the following lemma:

Lemma 5 (Identity Limiting State Distribution).

$$\vdash \forall\ \texttt{X p a n p}_0\ \texttt{i j}.\ \texttt{LRU\_model X p a n p}_0 \wedge \texttt{i} \in \texttt{[1, n]} \wedge \texttt{j} \in \texttt{[1, n]} \Rightarrow \lim_{\texttt{t}\to\infty}\mathbb{P}\{\texttt{x | X t x = i}\} = \lim_{\texttt{t}\to\infty}\mathbb{P}\{\texttt{x | X t x = j}\}$$

The HOL proof of the above lemma is based on Theorem 3 along with some arithmetic reasoning.

Theorem 11 implies that $\lim_{\texttt{t}\to\infty} p_i^{(n)}$ (for any tag i) is independent of its initial distribution and the position of the tagged page has an equal probability to be in any stack position. This means that any page is equally likely to be referenced in the long run. As a result, it concludes that this LRU stack specification does not cover the case of nonuniform page referencing behavior of programs. Thus, we have been able to formally verify the numerical methods result presented in [17].

The HOL code developed for the formalization and verification of the classified DTMCs is totally around 8000 lines and the proof script for verifying Theorems 11 and 10 is about 300 lines long, which are available at [9]. The ability to formally verify theorems involving classified Markovian models and the short script clearly indicates the usefulness of the formalization, presented in the earlier section of the paper, as without them the reasoning could not have been done in such a straightforward manner.

6 Conclusion

This paper presents the formalization of classified DTMCs along with some important prerequisites related to the formalization of DTMCs with finite state-space in a higher-order logic theorem prover. Our results facilitate the formal analysis of classified DTMCs and provides the foundations for formalizing more advanced concepts of Markov chain theory, like hidden Markov chains, Markov decision process and other useful properties. Due to the inherent soundness of theorem proving, our work guarantees to provide accurate results, which is a very useful feature while analyzing stationary behaviors of a system associated with safety or mission-critical systems. In order to illustrate the usefulness of the proposed approach, we formally analyzed a LRU stack model using the definitions of aperiodic and irreducible DTMC and their formally verified properties. Our results exactly matched the conclusion and the corresponding experimental results in [17], which ascertains the precise nature of the proposed approach.

The presented work opens the door to a new and very promising research direction, i.e., integrating HOL theorem proving in the domain of analyzing classified DTMCs. We are currently working on extending the set of formally verified properties regarding DTMCs and extending our work to time-inhomogeneous discrete-time Markov chains and Markov Decision Process (MDP), which will enable us to formally analyze a wider range of systems. We also plan to build upon the formalization of continuous random variables and statistical properties to formalize Continuous-Time Markov Chains (CTMC) to be able to formally reason about statistical characteristics of a broader scope of Markovian models.

References

1. Avi-Itzhak, B., Heyman, D.P.: Approximate Queuing Models for Multiprogramming Computer Systems. Operations Research 21(6), 1212–1230 (1973)
2. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
3. Bhattacharya, R.N., Waymire, E.C.: Stochastic Processes with Applications. John Wiley & Sons (1990)
4. Chung, K.L.: Markov chains with stationary transition probabilities. Springer (1960)
5. Häggström, O.: Finite Markov Chains and Algorithmic Applications. Cambridge University Press (2002)
6. Hasan, O.: Formal Probabilistic Analysis using Theorem Proving. PhD Thesis, Concordia University, Montreal, QC, Canada (2008)
7. Hasan, O., Tahar, S.: Reasoning about Conditional Probabilities in a Higher-Order-Logic Theorem Prover. Journal of Applied Logic 9(1), 23–40 (2011)
8. Hölzl, J., Nipkow, T.: Interactive verification of markov chains: Two distributed protocol case studies. In: Quantities in Formal Methods, EPTCS, pp. 17–31 (2012)
9. Liu, L. (2013), `http://hvg.ece.concordia.ca/code/hol/cdtmc/`
10. Liu, L., Hasan, O., Tahar, S.: Formalization of Finite-State Discrete-Time Markov Chains in HOL. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 90–104. Springer, Heidelberg (2011)
11. Mhamdi, T.: Information-Theoretic Analysis using Theorem Proving. PhD Thesis, Concordia University, Montreal, QC, Canada (2012)
12. Mhamdi, T., Hasan, O., Tahar, S.: On the Formalization of the Lebesgue Integration Theory in HOL. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 387–402. Springer, Heidelberg (2010)
13. Norris, J.R.: Markov Chains. Cambridge University Press (1999)
14. Parker, D.A.: Implementation of Symbolic Model Checking for Probabilitics Systems. PhD Thesis, University of Birmingham, Birmingham, UK (2002)
15. PRISM (2013), `http://www.prismmodelchecker.org`
16. Sen, K., Viswanathan, M., Agha, G.: VESTA: A Statistical Model-Checker and Analyzer for Probabilistic Systems. In: IEEE International Conference on the Quantitative Evaluation of Systems, pp. 251–252 (2005)
17. Shedler, G., Tung, C.: Locality in page reference strings. SIAM Journal on Computing 1(3), 218–241 (1972)
18. Trivedi, K.S.: Probability and Statistics with Reliability, Queuing, and Computer Science Applications. John Wiley & Sons (2002)
19. YMER (2013), `http://www.tempastic.org/ymer/`

Practical Probability: Applying pGCL to Lattice Scheduling

David Cock

NICTA and University of New South Wales
David.Cock@nicta.com.au

Abstract. Building on our published mechanisation of the probabilistic program logic pGCL we present a verified lattice scheduler, a standard covert-channel mitigation technique, employing randomisation as an elegant means of ensuring starvation-freeness. We show that this scheduler enforces probabilistic non-leakage, in addition to non-starvation. The refinement framework employed is compatible with that used in the L4.verified project, supporting our argument that full-scale verification of probabilistic security properties for realistic systems software is feasible.

1 Introduction

In this paper, we demonstrate that mechanically verifying realistic probabilistic software, with probabilistic properties, is feasible. Our 'realistic probabilistic program' is a hybrid probabilistic lattice-lottery scheduler, designed to mitigate information flow through a shared cache, while guaranteeing fairness and remaining simple and efficient. The probabilistic properties are: stochastic fairness — that the probability of starvation for any domain is zero, and non-leakage — that the distribution of observable outputs is independent of hidden inputs. Finally, we make our argument for feasibility by proving our results in a refinement framework compatible with the L4.verified [KEH$^+$09] proof stack, which established by refinement that the seL4 microkernel faithfully implements its specification. We are able to restrict probabilistic reasoning to small regions, allowing the remainder of the proof to proceed in a traditional manner.

We begin with an abstract, nondeterministic specification, which we refine iteratively. Our first refinement is to a probabilistic version, and then to a practical implementation based on lottery scheduling. We demonstrate that this refinement could be continued using the L4.verified results. Finally, we attach a hardware model, allowing us to demonstrate that we do in fact eliminate leakage through the cache.

We express our results using the probabilistic Guarded Command Language (pGCL) of McIver and Morgan [MM04], previously mechanised in HOL4 [HMM05]. This language has been applied in a practical context, namely analysing parts of the FireWire protocol [FS03]. We demonstrate that it is equally applicable to systems-level software. Our own mechanization of pGCL in Isabelle/HOL, specifically aimed at the lightweight integration of existing results, has been described previously [Coc12].

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 311–327, 2013.

We assume a lattice-based security policy, an established idea, motivated by institutional classification policies [DoD86], and formally treated by authors including Denning [Den76]. Lattice scheduling, due to Hu et al. [Hu92], arose from the VAX VMM project [KZB+91], and is intended to mitigate precisely the kind of leakage that we consider. Lottery scheduling is an established technique for efficient hierarchical allocation of execution time, introduced by Waldspurger et al. [WW94] Our approach uses it only as an elegant way to implement probabilistic scheduling: we do not take advantage of hierarchical resource distribution.

While the threat from cache-based channels has long been recognised, interest has been spurred by recent work demonstrating the feasibility of attacking cryptographic algorithms in a co-hosted system [Ber04, Per05]. Alternative mechanical approaches include that of Barthe et al. [BBCL12] Our work contrasts with this by incorporating probability, and interfacing with a large existing verification effort [KEH+09].

Many authors have analysed the leakage properties of scheduling algorithms [CM07,HN12,GKV11], some employing mechanical proof. Most existing analyses focus on leakage due to the actions of the scheduler itself, or due to the order of updates to shared variables. We are specifically concerned with mitigating a side channel, outside any explicitly shared state. The absence of unintended channels through explicit mechanisms in seL4 has already been established [MMB+12].

1.1 pGCL in Isabelle

We first summarise pGCL, noting small variations in syntax relative to the standard presentation. This summary is naturally incomplete, and the interested reader is directed to the aforementioned work of McIver and Morgan [MM04], and to our own previous summary of the mechanisation [Coc12].

Programs in pGCL have two interpretations: The first is as a probabilistic state transformer, taking a given starting state to one of several possible final states, with well-defined probability. The second interpretation is as an *expectation transformer*, mapping a real-valued function on final states (a post-expectation), to one on initial states (a pre-expectation). The *weakest pre-expectation* (wp) of a post-expectation, under a program, and evaluated at some initial state is the smallest *expected value* (minimised over demonic choices) of the post-expectation in the final state, if the program were to execute from the given initial state. For example, the weakest pre-expectation of the expression x, under the program

$$(x := 1 \;{}_{1/2}\!\oplus\; x := 0) \sqcap (x := 2 \;{}_{1/3}\!\oplus\; x := 1) \qquad ,$$

(where $a \;{}_{p}\!\oplus\; b$ is probabilistic choice and $a \sqcap b$ demonic) is

$$\min \left(\frac{1}{2} \times 1 + \frac{1}{2} \times 0\right) \left(\frac{1}{3} \times 2 + \frac{2}{3} \times 1\right) = \frac{1}{2} \qquad .$$

While our mechanisation is in terms of expectation transformers, the two interpretations are equivalent, and the forward transformer is generally more intuitive, giving the most straightforward way to visualise results.

Programs are constructed using several operators, including:

- Sequential composition: $a\,;;\,b$.
- Name binding: n is f in $a\ n$, where f is a function from state to value.
- Demonic choice: $x :\in S$, where S is a set-valued function, and x a variable (field) name.
- Probabilistic choice: $x :\in S$ at p, where p is a distribution over S.
- Finite repetition: $a^n = \underbrace{a\,;;\ldots;;\,a}_{n}$.
- Lifting from a non-probabilistic monad: Exec M.
- Applying a state transformer: Apply f.

While programs may operate on any state type, in practice we use Isabelle's *record types*: tuples with labelled fields, similar to C structures. The advantage is that with support from our mechanisation, we are able to use Isabelle field identifiers directly as pGCL variable names. For example we may write $x := v$, which is translated internally to Apply $\lambda s.\ x\_update\ (\lambda\_.\ v)\ s$, an Isabelle record update. This program could be applied to the following state, expressing a record of two fields, of types τ and μ:

$$\begin{aligned}\textbf{record}\ \text{state} = x &:: \tau \\ y &:: \mu\end{aligned}$$

The assertion language is shallowly embedded, and closely resembles the predicate-transformer semantics of Dijkstra's GCL [Dij75]. There are a few novel probabilistic constructions, including:

- Entailment: $P \Vdash Q = \forall s.\ P\ s \leq Q\ s$ (standard syntax $\Rrightarrow$).
- Conjunction: $P \mathbin{\&\&} Q = \lambda s.\ \max 0\ (P\ s + Q\ s - 1)$ (standard syntax &).
- Embedding: $«P» = \lambda s.$ if $P\ s$ then 1 else 0 (standard syntax $[P]$).

Probabilistic entailment is a straightforward generalisation of predicate entailment, $P \vdash Q \Leftrightarrow \forall s.\ P\ s \to Q\ s$, while embedding is simply syntactic sugar. The form of probabilistic conjunction is chosen for compatibility with its boolean equivalent i.e. $«P» \mathbin{\&\&} «Q» = «\lambda s.\ P\ s \wedge Q\ s»$. That we use this particular form (rather than, for example $P \mathbin{\&\&} Q = \lambda s.\ P\ s \times Q\ s$, which gives the same results on embedded predicates) is for technical reasons concerning the underlying semantic interpretation[1].

The following is an example specification in expectation-entailment style that illustrates the essential features of the logic:

$$«P» \mathbin{\&\&} (\lambda\_.\ p) \Vdash \text{wp}\ (a\,;;\,b)\ «Q»$$

[1] Briefly, the definition given is the only option that is *sub-linear*, a generalisation (to real-valued functions), and weakening, of the *linearity* condition required of expectation transformers in pure GCL. All sub-linear transformers are linear, and sub-linearity reduces to linearity in the case of embedded boolean predicates, but (for example) demonic choice, $a \sqcap b = \lambda s.\ \min\ (a\ s)\ (b\ s)$, is not linear if a or b take values other than 0 and 1. All pGCL primitives are sub-linear, this being the healthiness condition for transformers. For further details, see McIver & Morgan [MM04].

This states that from any initial state satisfying P, after executing a followed by b, we reach a state satisfying Q with probability *at least* p.

Conventions. For simplicity, we employ the following conventions throughout: Unbound variables are implicitly universally quantified, and all expectations are non-negative, and bounded by 1.

2 Security Policies and Covert Channels

We consider a hierarchically partitioned system, as depicted in Figure 1. Here, all data is classified with one (or both) of the labels A and B. An agent (or program) may be cleared to process one, both, or neither of these, giving rise to 4 *clearance domains*: 1 for A only, 2 for B only, 3 for both and $\bot$ for neither. Our goal is to ensure that information derived from labelled data can only flow into a domain cleared to process it. The formulation of access control policies for such systems, encompassing explicit channels, is a well-studied problem [Den76]. This work is concerned with formalising implementation techniques to prevent leakage through unintended, implicit channels (either covert- or side-channels).

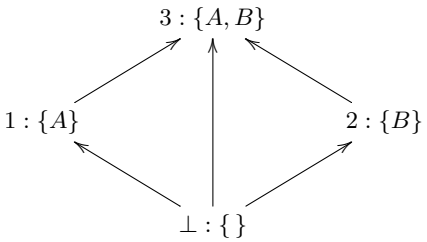


Fig. 1. The classification/clearance lattice

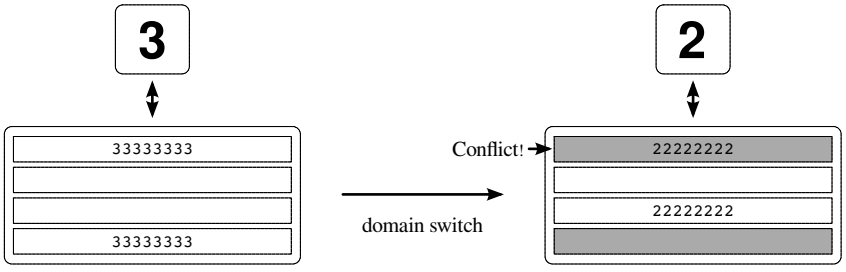


Fig. 2. The cache-contention channel

Figure 2 outlines such a channel, exploiting *cache contention*. The cache is represented by an array of cells (lines), and 3 and 2 are two of our supposedly isolated domains. We ignore the associativity of the cache (one cell may in fact hold several lines), as it only serves to reduce contention.

As domain 3 executes and accesses memory, it gradually fills the cache with its own data. On switching to domain 2, domain 3's data remains in the cache, but is now inaccessible (the greyed-out cells). This access control is enforced by the hardware. As 2 starts to fill the cache, it may eventually attempt to store a value in one of the grey squares, encountering a *conflict*, as indicated.

When a conflict occurs, the cache silently writes (cleans) the old value (domain 3's) into main memory, before storing domain 2's new value[2]. This process is in principle invisible to domain 2. However, if 2 is able to measure its own execution time, the delay caused by writing (cleaning) the cache line to memory can be detected. Domain 2 can thus infer which cache lines 3 has accessed, in violation of the security policy.

The leakage is dramatic: On a uniprocessor, where domains cannot execute concurrently, the bandwidth tops 10kb/s, while on a multiprocessor, bandwidths in excess of 1Mb/s are easily achieved.

3 Countermeasures through Refinement

The simplest countermeasure to the cache channel is to flush the cache on every domain switch, returning it to a known secure state. This is, however, a very expensive option: A modern processor, for example an Intel Xeon E7-8870, might have a 30MiB cache, taking $\approx 2.5 \times 10^6$ cycles to refill (at the peak theoretical bandwidth of the memory subsystem), or 89% of the 2.8×10^6 cycles per preemption interval at 1000Hz.

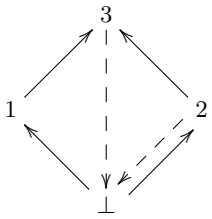


Fig. 3. The scheduling graph, S

A simple optimisation [Hu92] is to clear the cache only when essential. Considering our partitioned system, it is acceptable to permit leakage from a domain to any other domain whose clearance includes that of the first. Thus it is only necessary to flush when decreasing clearance level. This is the essence of *lattice*

[2] In a write-back cache with write-allocate. Read contention occurs in all caches.

scheduling: Transition upward in the classification lattice for as long as possible, before finally starting again at the bottom, employing countermeasures to protect the downward transition.

To implement this, we construct the *scheduling graph* in Figure 3; consistent with the classification graph in Figure 1. The scheduling graph gives valid domain transitions for the system, and contains only edges from the classification graph, or transitions to the *downgrader*, $\bot$. These latter are emphasised with a dashed arrow. In the implementation, the shared cache must be flushed on entering the downgrader. We omit the edges from $\bot$ to 3, and from 1 to $\bot$, to emphasise that not all edges need be included.

The conditions on the scheduling graph (modelled as a relation) are captured as assumptions on S (encapsulated within an Isabelle locale), with the most important being downgrading:

Lemma 1 (Downgrading). *If S allows a downward transition, it is to the downgrader, $\bot$:*

$$\frac{(c, n) \in S \quad \mathit{clearance}\ c \not\subseteq \mathit{clearance}\ n}{n = \bot}$$

We specify the scheduler nondeterministically over the valid transitions from the current domain, using the unconstrained demonic choice operator:

record stateA = current_domain :: *dom_id*
scheduleS =
  c is current_domain in
  current_domain :$\in$ (λ_. $\{n.\ (c, n) \in S\}$)

3.1 A Randomised Scheduler

The classically nondeterministic specification of scheduleS, together with the downgrading property, capture the requirement that all downward transitions pass through the downgrader. As a practical specification however, it has a disadvantage: it allows starvation. A refinement of this specification is free to

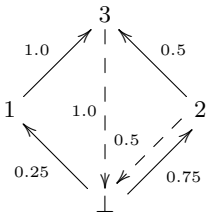


Fig. 4. The transition graph, T

follow any trace within the graph, for example $(\bot, 2, \bot, 2, \ldots)$, never scheduling domain 3.

We could extend the specification to guarantee starvation freeness, by dictating its behaviour over traces in a modal logic. This would risk obscuring the present simplicity of the specification, and would require a more complex implementation, needing to take into account more than just the current domain.

Randomisation provides an elegant alternative: By assigning a probability to each edge in Figure 3, we produce the *transition graph* in Figure 4. We only require that the outgoing probabilities from each node sum to 1, and that any transition with non-zero probability appears as an edge in Figure 3. Implementation remains simple, needing only to consider the current state in choosing a transition. More importantly, with appropriately chosen transition probabilities, the probability of starvation can be made zero. We specify the new scheduler using the probabilistic choice operator:

$$\begin{aligned}&\text{scheduleT} = \\ &\quad c \text{ is current\_domain in} \\ &\quad \text{current\_domain} :\in (\lambda\_.\ \{\bot, 1, 2, 3\} \text{ at } (\lambda\_\ n.\ T\ (c, n))\end{aligned}$$

The scheduler is now a Markov process, with T fixing its transition rule. Under the appropriate conditions (strong-connectedness, or positive recurrence interval for all states), there exists an asymptotic equilibrium distribution. These conditions are satisfied by T, and thus in addition to avoiding starvation, the randomised lattice scheduler guarantees statistical fairness, over the long run.

3.2 Program Refinement and Starvation-Freedom

In order to eventually show non-leakage, we need to demonstrate that the downgrading property is also shared by scheduleT. We do so by establishing that scheduleT is a *probabilistic refinement* of scheduleS.

Definition 1. *Program b* refines *program a, written $a \sqsubseteq b$, exactly when all expectation-entailments on a also hold on b:*

$$\frac{P \Vdash wp\ a\ Q}{P \Vdash wp\ b\ Q}$$

Lemma 2. *The transition scheduler refines the lattice scheduler:*

$$scheduleS \sqsubseteq scheduleT$$

Note that, in the terminology of pGCL, the specification of scheduleT is completely 'deterministic', referring to the absence of demonic nondeterminism. This terminology makes sense in light of the refinement order: Demonic nondeterminism can be restricted by refinement, whereas probabilistic choice cannot. Once a

specification is fully probabilistic, it is maximal in the refinement lattice, and one can take it no further. This implies that any further refinement is, in fact, semantic equivalence. We make use of this fact shortly, as a shortcut to establishing program correspondence.

Having fixed transition probabilities, we can establish non-starvation. Proceeding in stages, we first show that starting in *any* domain, the probability of ending in domain $\bot$ after 4 steps is at least 1/64:

$$(\text{in\_dom } d_i) \,\&\&\, \left(\lambda\_.\ \frac{1}{64}\right) \Vdash \text{wp scheduleT}^4\ (\text{in\_dom } \bot)$$

where

$$\text{in\_dom } d \quad \leftrightarrow \quad \text{«}\lambda s.\ \text{current\_domain } s = d\text{»}$$

We further establish that from domain $\bot$, after a further 4 steps, there is a non-zero probability of ending in any given final domain:

$$(\text{in\_dom } \bot) \,\&\&\, \left(\lambda\_.\ \frac{1}{64}\right) \Vdash \text{wp scheduleT}^4\ (\text{in\_dom } d_f)$$

Combining these, we have:

$$\left(\lambda\_.\ \frac{1}{4096}\right) \Vdash \text{wp scheduleT}^8\ (\text{in\_dom } d_f) \tag{1}$$

Finally:

Lemma 3 (Non-starvation). *Taking at least 8 steps from* any *initial domain, we reach* any *final domain with non-zero probability:*

$$\forall s.\ 0 < \mathit{wp}\ \mathit{scheduleT}^{8+n}\ (\mathit{in\_dom}\ d_f)\ s$$

Proof. By induction on n. Equation 1 establishes the result for $n = 0$. By inspection of Figure 4, we see that every domain is reachable in one step, and with non-zero probability, from at least one other, and thus if all domains are reachable after n steps then all are reachable after $n + 1$. □

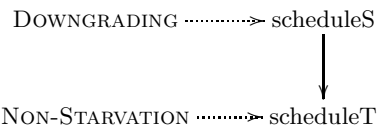


Fig. 5. First Refinement Diagram

Figure 5 summarises these results. We have downgrading for scheduleS by assumption, and non-starvation for the probabilistic scheduleT, as indicated by the dotted arrows. Refinement is depicted as a solid arrow. The arrow directions summarise the compositionality of results: composing with refinement, downgrading also holds for scheduleT, but non-starvation does not hold for scheduleS.

3.3 Data Refinement and the Lottery Scheduler

It is not sufficient to have an elegant specification, unless that specification can be practically implemented. Therefore we implement our randomised lattice scheduler as a *lottery scheduler*. We only require the assumption of randomness for a single operation: drawing a ticket.

We extend the abstract state with a *lottery* for each domain. Every possible successor domain holds a certain set of tickets, given by the function 'lottery'. To transition, the scheduler draws a ticket (32 word) and consults the table to choose a successor. To emphasise that the probabilistic component can be isolated, and to demonstrate compatibility with our existing framework, we divide the implementation into a core, in the nondeterministic state monad [CKS08], which is then lifted into pGCL using the Exec operator, allowing us to employ probabilistic choice. Both scheduleC and scheduleM operate on the same state space: stateC. The syntax $r(\!|x := y|\!)$ is an Isabelle record update, assigning value y to field x of record r.

$$
\begin{aligned}
\textbf{record}\ \text{domain} &= \text{lottery} :: 32\ \text{word} \Rightarrow \mathit{dom\_id} \\
\textbf{record}\ \text{stateC} &= \text{current\_domain} :: \mathit{dom\_id} \\
&\quad\ \text{domains} :: \mathit{dom\_id} \Rightarrow \text{domain} \\
\text{scheduleM}\ t &= \text{do}\ c \leftarrow \text{gets current\_domain} \\
&\qquad\ \mathit{dl} \leftarrow \text{gets domains} \\
&\qquad\ \text{let}\ n = \text{lottery}\ (\mathit{dl}\ c)\ t\ \text{in} \\
&\qquad\ \text{modify}\ (\lambda s.\ s(\!|\text{current\_domain} := n|\!)) \\
&\quad\ \text{od} \\
\text{scheduleC} &= t\ \text{from}\ (\lambda s.\ \text{UNIV})\ \text{at}\ 2^{-32}\ \text{in} \\
&\quad\ \text{Exec}\ (\text{scheduleM}\ t)
\end{aligned}
$$

Having moved to a new state space, we cannot have direct program refinement between scheduleT and scheduleC. Noting, however, that the abstract state can be recovered from the concrete by projection, we instead have *(projective) probabilistic data refinement*:

Definition 2 (Probabilistic Data Refinement). *Program b, on state type σ, refines program a, state τ, given precondition $G : \sigma \to Bool$ and under projection $\theta : \sigma \to \tau$, written $a \sqsubseteq_{G,\theta} b$, exactly when any expectation entailment on a implies the same for b, on the projected state and with a guarded pre-expectation:*

$$\frac{P \Vdash wp\ a\ Q}{«G» \,\&\&\, (P \circ \theta) \Vdash wp\ b\ (Q \circ \theta)}$$

Lemma 4. *Let $\|S\|$ be the cardinal measure (element count) of set S.*
Under condition LR, that 'lottery' reflects the transition matrix,

$$T\ (c, n) = 2^{-32} \|\{t.\ lottery\ (domains\ s\ c)\ t = n\}\|$$

then under projection ϕ, which extracts the current domain,

$$current\_domain\ (\phi\ s) = current\_domain\ s$$

scheduleC is a data refinement of scheduleT:

$$scheduleT \sqsubseteq_{LR,\phi} scheduleC$$

3.4 Probabilistic Correspondence

As mentioned in Section 3.2, scheduleT is maximal in the refinement order, and thus any refinement is an equivalence. This is *probabilistic correspondence*:

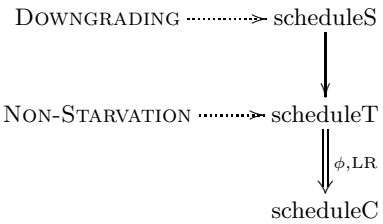


Fig. 6. Second Refinement Diagram

Definition 3 (Probabilistic Correspondence). *Programs a and b are said to be in* probabilistic correspondence*, pcorres θ G a b, given condition G and under projection θ if, for any post-expectation Q, the guarded pre-expectations coincide:*

$$«G» \mathbin{\&\&} (wp\ a\ Q \circ \theta) = «G» \mathbin{\&\&} wp\ b\ (Q \circ \theta)$$

Probabilistic correspondence is guarded equality on distributions: From an initial state satisfying G, a and b establish Q with equal probability. The advantage of detouring via refinement, rather than directly showing correspondence, is that the proof is simpler; the next result follows directly from Lemma 4:

Lemma 5. *The specifications scheduleT and scheduleC correspond given condition LR and under projection ϕ:*

$$pcorres\ \phi\ LR\ scheduleT\ scheduleC$$

This extends Figure 5 to Figure 6, with correspondence indicated by the double arrow. As correspondence implies refinement, both downgrading and non-starvation hold for scheduleC, as implied by the arrows. Properties represented by a single dotted arrow (e.g. downgrading), are preserved by both refinement (single arrow) and correspondence (double arrow).

3.5 Proof Reuse: Composing with seL4

Our argument for the feasibility of this approach rests on the compatibility of probabilistic correspondence with its non-probabilistic equivalent at the heart of the L4.verified proof. We have previously demonstrated the ease with which monadic specifications, in the style of seL4, can be re-used in a probabilistic setting [Coc12], automatically lifting Hoare triples to probabilistic predicate entailment relations. With the following result we go further, and lift the bulk of the refinement stack. The predicate corres_underlying in the following lemma is the fundamental definition which underlies the refinement results at all levels of the L4.verified proof [CKS08]. Here, we need only note that this is the form of the top-level theorem[3].

Lemma 6 (Lifting Correspondence). *Given correspondence between monadic programs M and M', with precondition G and projective state relation ϕ,*

$$corres\_underlying\ \{(s, s').\ s = \phi\ s'\}\ True\ rrel\ G\ (G \circ \phi)\ M\ M'$$

where M does not fail given G,

$$no\_fail\ G\ M$$

and neither diverges without failing,

$$empty\_fail\ M \quad empty\_fail\ M'$$

and that M is deterministic on the image of the projection,

$$\forall s.\ \exists (r, s').\ M\ (\phi\ s) = \{(False, (r, s'))\}$$

[3] Briefly, corres_underlying *srel nf rrel G G′ m m′* is defined as:

$$\forall (s, s') \in srel.\ G\ s \wedge G'\ s' \rightarrow (\forall (r', t') \in \text{fst}\ (m'\ s').\ \exists (r, t) \in \text{fst}\ (m\ s).\ (t, t') \in srel \wedge \text{rrel}\ r\ r' \wedge (nf \rightarrow \neg \text{snd}\ (m'\ s')))$$

Where guards G and G' hold on initial states s and s' satisfying state relation *srel*, for any pair of (result, final state) obtained by executing m', there exists a corresponding pair obtainable by executing m. If the non-failure flag, *nf* is set, then the predicate additionally asserts that m' does not fail.

We use the predicate with a projective relation derived from ϕ, no failure, an arbitrary result relation, and a concrete guard which is the anti-projection of the abstract guard $(G \circ \phi)$.

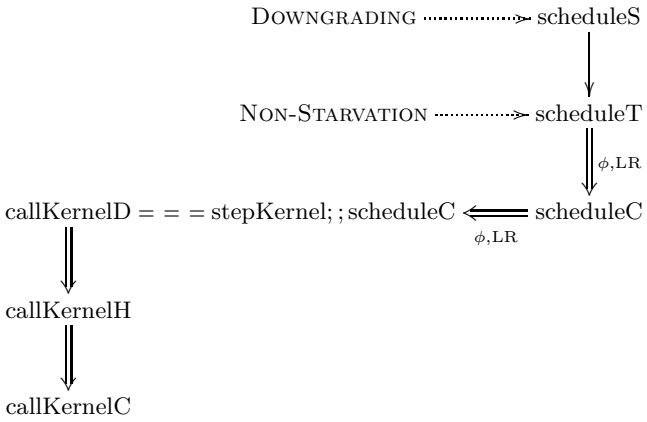


Fig. 7. Composed Refinement Diagram

then we have probabilistic *correspondence between their lifted counterparts:*

$$pcorres\ \phi\ (G \circ \phi)\ (Exec\ M)\ (Exec\ M')$$

Note that the final assumption is exactly the determinism[4] condition that we previously established for scheduleT, restricted to the components of interest. M is free to behave nondeterministically on components which are masked by the projection.

Thus we may compose our probabilistic results with the deterministic levels of the L4.verified proof (the executable, or more recent deterministic abstract [MM12], specification). For the problem at hand, it is only necessary to make a few assumptions on the kernel:

Lemma 7. *If the kernel preserves the lottery relation,*

$$\{\!|LR|\!\}\ stepKernel\ \{\!|\lambda\_.\ LR|\!\}$$

and the current domain,

$$\{\!|\lambda s.\ CD\ s = d|\!\}\ stepKernel\ \{\!|\lambda\_\ s.\ CD\ s = d|\!\}$$

and is total,

$$no\_fail\ \top\ stepKernel \quad empty\_fail\ stepKernel$$

[4] Determinism gives us correspondence, rather than just refinement. Consider monads A and A', and variable $x : \mathbb{N}$, preserved by projection ϕ_A. Let $A\ s$ be nondeterministic, giving either$s(\!|x := x\ s+1|\!)$ or $s(\!|x := x\ s+2|\!)$, while $A'\ s$ is deterministic, giving $s(\!|x := x\ s+2|\!)$. All behaviours of A' are included in A, and thus corres_underlying holds. However, wp $A\ x = \lambda s.\ x\ s+1$ whereas wp $A'\ (x \circ \phi_A) = \lambda s.\ x\ s+2$: a refinement, but not correspondence. As previously mentioned, if A were deterministic then by maximality, this refinement would be correspondence.

then with the concrete scheduler, it refines the transition scheduler:

$$scheduleT \sqsubseteq_{LR,\phi} stepKernel;;scheduleC$$

With this (again using refinement to show correspondence), Figure 6 becomes Figure 7, now including the lifted kernel. The L4.verified refinement stack is depicted on the left to indicate how the results would compose, to take our result down to the real, executable kernel. Here callKernelD is the deterministic refinement of original abstract specification of seL4, callKernelH is the executable model derived from the Haskell prototype, and callKernelC is the concrete model, comprising the final C and assembly language implementation.

So far, we have only shown that our results are *compatible*: we do not yet have a mechanised proof. The remaining results are the first two assumptions of Lemma 7, which will hold by construction as the existing kernel clearly cannot modify the additional scheduler state, and the fact that the state relation is projective: that is, that the abstract state is uniquely recoverable from the concrete state. This is the intended behaviour of the state relation, and we have no reason to suspect that this result will not hold.

3.6 Non-leakage with a Concrete Machine Model

Our ultimate goal is to show the absence of information leakage via shared state (specifically the processor cache), and so we extend our scheduler with a simple hardware model. We model a private state per domain (memory), and a single shared state (cache):

$$\begin{aligned}\textbf{record}\ (sh, pr)\ \text{machine} = \ &\text{private} :: dom\_id \Rightarrow pr \\ &\text{shared} :: sh\end{aligned}$$

The action of a domain is modelled by the underspecified function runDom :: $sh \times pr \Rightarrow sh \times pr$, acting on both the current domain's private state and the shared state. Only the action of domain $\bot$ is specified, and then only on the shared state, resetting it.

The model exposes the essential information-flow characteristics of the cache channel, as illustrated by Figure 8. Initially, the states associated with domain 3 (black) and 2 (grey) are isolated. After a single step, domain 3's influence propagates to the cache (S), but as yet no other private state has been affected. It is only after the second step that influence propagates to 2's private state, it and the cache now being influenced by both 2 and 3's initial states. As this mixing of private states cannot occur in less than 2 steps, and may take an unbounded time (2's state cannot be influenced until 2 is scheduled), we cannot formulate a one-step security property. Instead we have a trace property, enforcing that after any number of steps, the distribution of outcomes visible to a low observer is independent of any initial high state, a form of *probabilistic non-leakage* [vO04]:

Lemma 8 (Non-leakage). *If the clearance of domain h is not entirely contained within that of domain l,*

$$clearance\ h \not\subseteq clearance\ l$$

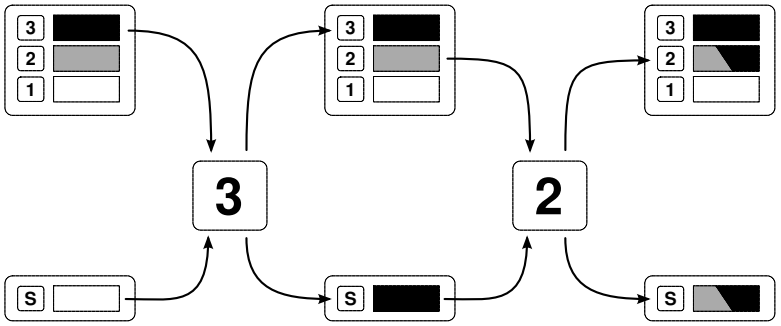


Fig. 8. A schematic depiction of flow from 3 to 2, via shared state S

then any function of the state after execution, which depends only on elements within l's clearance,

$$Q \circ mask\ l$$

is invariant under modifications to h's private state (as represented by replace):

$$wp\ (runDom;;scheduleT)^n\ (Q \circ mask) = (wp\ (runDom;;scheduleT)^n\ (Q \circ mask)) \circ (replace\ h\ p)$$

We also have correspondence between scheduleT and runDom;;scheduleC:

Lemma 9. *Assuming that the lottery relation LR holds, then under projection ψ, which drops the machine state, we have the correspondence:*

$$pcorres\ LR\ \psi\ (scheduleT)\ (runDom;;scheduleT)$$

and thus by compositionality,

$$pcorres\ LR\ (\psi \circ \phi)\ (scheduleT)\ (runDom;;scheduleC)$$

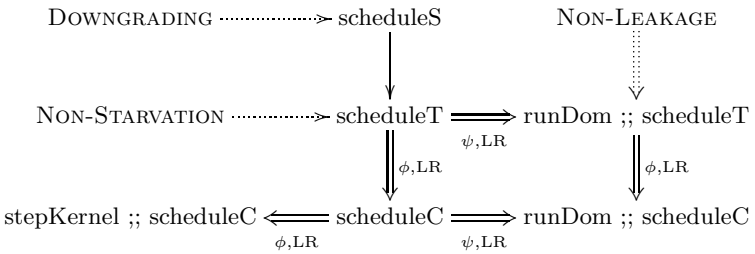


Fig. 9. The Complete Refinement Diagram

Therefore, finally, we have all three results: downgrading, non-starvation and non-leakage, on the concrete lottery scheduler composed with the hardware model, as depicted in Figure 9. Here, non-leakage is shown using a double dotted arrow to emphasise that it is only preserved by correspondence, and not by refinement.

4 Conclusions

We have presented a hybrid probabilistic lattice-lottery scheduler, which allows efficient mitigation of the cache channel, while simultaneously guaranteeing non-starvation. Working in pGCL, our system is produced by iterative refinement, supplemented by mechanical proof. This demonstrates that given adequate tool support (namely Isabelle/HOL and our mechanisation of pGCL), refinement-driven development and verification of realistic *probabilistic* systems software is no more difficult than the existing non-probabilistic case. We have shown that our refinement framework is compatible with that of the L4.verified project, and set out the steps necessary to combine this work with a system such as seL4, giving a mechanical proof down to a real system of probabilistic top-level properties. Above all, we argue that verifying probabilistic security properties on realistic systems software is entirely feasible with current technology.

5 Ongoing and Future Work

We have established non-starvation in Lemma 3 as a property of finite traces (of length at least 8). While weaker than this, it would be nice to derive the standard formulation of non-starvation: that any given domain will eventually be scheduled, or $\forall d.\ \Diamond(\text{current\_domain} = d)$ in the syntax of a boolean modal logic. In our case, of course, the result must necessarily be probabilistic: that it is 'almost certain' that any domain is eventually scheduled. We have already partially mechanised the quantitative temporal logic, qTL, of Morgan and McIver [MM99], which allows us to express this result as $\forall d.\ \Diamond(\text{current\_domain} = d) = \mathbf{1}$, with boolean predicates generalised to real-valued expectations, as for pGCL. We have so far managed to feed our unmodified pGCL results into qTL, and anticipate that these results will appear in a forthcoming work.

Progress on the assumptions of Lemma 7, required for the connection to seL4, is ongoing. In a separate work, currently under submission, our colleagues Daum & Billing have shown that the seL4 state relation is indeed projective, satisfying the implicit assumption. Formally proving the explicit assumptions (lottery relation and current domain preservation) presents no theoretical challenges, simply requiring a large but trivial proof. Integrating the projectivity result should be similarly straightforward. The more interesting question is what form that the final top-level statement should take to cleanly integrate the probabilistic and classical properties of seL4, and is the subject of ongoing research.

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.

The author is also indebted to the anonymous reviewers for their insightful and constructive feedback, and to Toby Murray and Matthias Daum for reviewing draft copies of this paper.

References

[BBCL12] Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Cache-leakage resilient os isolation in an idealized model of virtualization. In: 25th Comp. Security Foundations WS, pp. 186–197 (2012)

[Ber04] Bernstein, D.J.: Cache-timing attacks on AES (2004)

[CKS08] Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)

[CM07] Chen, H., Malacaria, P.: Quantitative analysis of leakage for multi-threaded programs. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, pp. 31–40. ACM, New York (2007)

[Coc12] Cock, D.: Verifying probabilistic correctness in isabelle with pGCL. In: Systems Software Verification, Sydney, Australia, p. 10 (November 2012)

[Den76] Denning, D.E.: A lattice model of secure information flow. CACM 19, 236–242 (1976)

[Dij75] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. CACM 18(8), 453–457 (1975)

[DoD86] US Department of Defence. Trusted Computer System Evaluation Criteria, DoD 5200.28-STD (1986)

[FS03] Fidge, C., Shankland, C.: But what if i don't want to wait forever? Formal Aspects of Computing 14, 281–294 (2003)

[GKV11] Gong, X., Kiyavash, N., Venkitasubramaniam, P.: Information theoretic analysis of side channel information leakage in FCFS schedulers. In: 2011 IEEE International Symposium on Information Theory Proceedings (ISIT), pp. 1255–1259 (August 2011)

[HMM05] Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in HOL. Theoretical Computer Science 346(1), 96–112 (2005)

[HN12] Huisman, M., Ngo, T.M.: Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 178–195. Springer, Heidelberg (2012)

[Hu92] Hu, W.M.: Lattice scheduling and covert channels. In: IEEE Symp. Security & Privacy, pp. 52–61 (1992)

[KEH+09] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT, USA, pp. 207–220. ACM (2009)

[KZB+91] Karger, P.A., Zurko, M.E., Bonin, D.W., Mason, A.H., Kahn, C.E.: A retrospective on the VAX VMM security kernel. Trans. Softw. Engin. 17(11), 1147–1165 (1991)

[MM99] Morgan, C., Mciver, A.K.: An expectation-based model for probabilistic temporal logic. Logic Journal of the IGPL 7, 779–804 (1999)

[MM04] McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer (2004)

[MM12] Matichuk, D., Murray, T.: Extensible specifications for automatic re-use of specifications and proofs. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 333–341. Springer, Heidelberg (2012)

[MMB+12] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 126–142. Springer, Heidelberg (2012)

[Per05] Percival, C.: Cache missing for fun and profit. In: BSDCan 2005 (2005)

[vO04] von Oheimb, D.: Information flow control revisited: Noninfluence = noninterference + nonleakage. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (eds.) ESORICS 2004. LNCS, vol. 3193, pp. 225–243. Springer, Heidelberg (2004)

[WW94] Waldspurger, C.A., Weihl, W.E.: Lottery scheduling: Flexible proportional-share resource management. In: 1st OSDI, Monterey, CA, USA, pp. 1–11 (November 1994)

Adjustable References

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. Even when programming purely mathematical functions, mutable state is often necessary to achieve good performance, as it underlies important optimisations such as path compression in union-find algorithms and memoization. Nevertheless, verified programs rarely use mutable state because of its substantial verification cost: one must either commit to a deep embedding or follow a monadic style of programming. To avoid this cost, we propose using adjustable state instead. More concretely, we extend Coq with a type of adjustable references, which are like ML references, except that the stored values are only partially observable and updatable only to values that are observationally indistinguishable from the old ones.

1 Introduction

Interactive proof assistants (Coq [3], Isabelle [13], HOL [11], etc.) are generally very good at reasoning about terminating purely functional computations, as these are just mathematical functions, which the provers support natively. In contrast, however, they are not so good at reasoning about real computations, which are not necessarily always terminating or purely functional. Such computations are not supported natively, but have to be encoded in some non-trivial way. This is a problem, because while requiring termination or productiveness for real computations may seem a reasonable restriction, disallowing mutable state is not at all reasonable, as many programs use mutable state for efficiency.

There are essentially two approaches for encoding stateful computations in an interactive proof assistant, both of which have their limitations.

First, in the *deep embedding approach* (e.g., [1,4,14]), impure computations are represented as terms of a custom data structure. Then, to reason about such terms, the user effectively has to build a specialized theorem prover for such terms. This is at the same time a rather challenging and a rather mundane task, as one cannot reuse much of the infrastructure of the interactive theorem prover, but may rather have to (re)implement standard features such as variable binding (cf. the PoplMark challenge [2]) or equating terms up to reduction.

Alternatively, there is the *monadic approach* (e.g., [8,9]), where imperative code is written in a monadic style against the state monad. This approach is usually preferable to the deep embedding approach as it reuses the interactive prover's infrastructure. It is nevertheless problematic because one cannot use global state to optimise purely functional code without forcing all the code depending on it to be written in a monadic style. Consider, for example, a mathematical function of type $A \to B$. If we apply the memoization optimisation

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 328–337, 2013.

(remembering the results of earlier function invocations in a hashtable so as to avoid recomputation), we will get a function of type $A \to \mathsf{StateMonad}\, B$. Thus, the memoized function, although intuitively equivalent to the original one, has a different type and can, therefore, no longer be used in arbitrary contexts, but only within contexts expecting monadic stateful computations as arguments.

To avoid these problems, we introduce a restricted form of mutable state, which we call *adjustable references*, that can be used freely inside pure computations (see §2). Similar to mutable references in ML, adjustable references store some internal value, but unlike ML references, the values stored cannot be arbitrarily updated. They can only be 'adjusted' so that the represented value remains the same, but perhaps affecting the cost of returning/computing it. From a verification point of view, we have to show that every adjustment is effectively an identity function. Then, when reasoning about code using adjustments, we can simply ignore the adjustments. In the extracted implementations, however, we perform the adjustments, as these can be very beneficial in terms of performance.

Adjustable references allow us to implement advanced persistent data structures by enabling imperative updates provided that they affect only the efficiency of the data structure accessor methods and not their results. As examples, we present two standard imperative optimisations that can be easily expressed using adjustable references: (*i*) memoization of function calls (see §3) and (*ii*) path compression of the union-find representation tree (see §4). The formal development associated with this paper can be found at the URL below:

`http://www.mpi-sws.org/~viktor/arefs/`

2 Adjustable References

In this section, we present an axiomatisation of adjustable references in Coq [3] and two implementations: (*i*) a purely-functional one in Coq that ensures the logical consistency of the axioms, and (*ii*) an efficient imperative one in OCaml. While adjustable references can be encoded in other proof assistants, we remark that our axiomatisation uses dependent types.

Our axiomatisation uses the Coq **Axiom** `x : T` and **Parameter** `x : T` declarations, which extend the context axiomatically with a new global constant, `x`, having the given type, `T`. (The former is typically used for propositions, and the latter for other types.)

Adjustable references internally store values of a representation type, R, but do not allow direct read-access to those values. The values are instead observable only at a possibly different type, T. Each adjustable reference type, `aref` f, thus has an associated observation function, $f : R \to T$, mapping values of its internal representation type to ones of its observation type.

Parameter `aref : ∀R T, (R → T) → Type.`

Adjustable references have a canonical constructor, `aref_val` $f\, v$, which creates a new reference of type `aref` f holding the internal value v. Our first axiom insists that every adjustable reference contains some internal value.

```
Parameter aref_val : ∀R T (f: R → T), R → aref f.
Axiom aref_inh :
      ∀ R T (f: R → T) (r: aref f), ∃v, r = aref_val f v.
```

Further, we have an operation, `aref_get` r, which reads the adjustable reference r and returns its 'external' value—namely, the result of applying the observation function to its internal value.

```
Parameter aref_get : ∀R T (f: R → T), aref f → T.
Axiom aref_get_val :
      ∀ R T (f: R → T) v, aref_get (aref_val f v) = f v.
```

Next, we need a way of adjusting the contents of the reference cell. Naively, one might think of axiomatising an operation of the following type:

$$\forall R, T.\ \forall f : R \to T.\ \forall r : \texttt{aref}\, f.\ \forall v : R.\ f(v) = \texttt{aref\_get}\,(r) \to \mathsf{unit}\,.$$

That is, we should be able to replace the r's current internal value with any value, v, that is observationally indistinguishable: i.e., $f(v) = \texttt{aref\_get}\,(r)$.

There are, however, three problems with this type.

1. First, Coq does not fix an evaluation order. In particular, the evaluation of $e_1; e_2$ (standing for **let** $x = e_1$ **in** e_2 where $x \notin \mathsf{fv}(e_2)$) may run e_1 and e_2 in any order. Moreover, it may not even evaluate e_1 at all since its value is never used. Coq's extraction [7], for example, does exactly so. It transforms $e_1; e_2$ into e_2, thereby erasing any 'adjustments' made in e_1. This problem on its own is not insurmountable, as we can force the evaluation of e_1 by redefining $e_1; e_2$ to mean **match** e_1 **with** $\mathsf{tt} \Rightarrow e_2$ **end**.
2. Second, the type above is overly restrictive. It requires the new internal value to be provided directly, thereby allowing it to depend only on the observable values of adjustable reference cells. In particular, it cannot depend on the previous internal value of the same reference cell. This is problematic if we want to use adjustable references to store some sort of cache. A typical adjustment of such a reference would be to extend the cache with one more entry, but to do so, the adjusted cache should clearly be able to depend on the old cache.
3. Third, unless the externally observable type, T, is a function type, $A \to B$, there is not much point in using adjustable references: we can just as simply calculate $f(v)$ at creation time, and then store the result in an immutable reference. For a function type, however, one may gradually refine the stored internal value each time the function is called, and therefore potentially improve the performance of future function calls.

For these reasons, we provide a combined adjustment read operation customized to the case of $T = (A \to B)$.

```
Parameter aref_getu :
  ∀ R A B (f : R → A → B) (upd : R → A → R * B)
      (PF: ∀x a, f (fst (upd x a)) = f x)
      (PF': ∀x a, snd (upd x a) = f x a),
    aref f → A → B.
```

Here, we must provide a function, `upd`, which when given the current internal representation and the function argument returns a pair consisting of the new representation and the result of calling f with these two arguments.

Adjusting reads have a remarkably simple axiomatisation: logically they are exactly the same as normal reads!

```
Axiom aref_getuE : ∀R A B (f : R → A → B) upd PF PF',
                   aref_getu upd PF PF' = aref_get (f:= f).
```

Coq experts will note that the proof obligations of `aref_getu` use the normal propositional Leibniz equality, and may be concerned that these proof obligations will be difficult to satisfy if B is itself a function type. This apparent problem, however, is not very serious as there are multiple ways around it. For example, one may assume the functional extensionality axiom when doing such proofs, as the axiom is consistent with Coq and its use will moreover not impact execution, since extraction erases these arguments. Alternatively, one may uncurry the function type, T, or simply extend the definition to one expecting a curried function of n arguments; i.e., with $T = (A_1 \to \ldots \to A_n \to B)$.

When adjusting the value of a reference cell, we have seen that it is useful for the new internal value to depend on the old internal value of the cell. Something similar holds for allocation of new adjustable reference cells: it is useful for the new internal value to depend on the internal value of some old reference cell. Thus, we also assume the following operation:

```
Parameter aref_new :
  ∀ R1 T1 (f1: R1 → T1) (r: aref f1)
    R2 T2 (f2 : R2 → T2) (g: ∀v, aref_get r = f1 v → R2)
    (PF: ∀x pfx y pfy, f2 (g x pfx) = f2 (g y pfy)),
    aref f2.
Axiom aref_new_val :
  ∀ R1 T1 (f1: R1 → T1) v R2 T2 (f2: R2 → T2) g PF,
    aref_new (aref_val f1 v) g PF
    = aref_val f2 (g v (aref_get_val f1 v)).
```

Having the internal value of a new reference cell depend on that of one old reference cell suffices for our examples, but it may easily be extended to making the internal value of the new cell depend on the internal values of a list of existing reference cells.

2.1 Logical Consistency of Adjustable References

In order to show that the new axioms about adjustable references are logically consistent, we present a very naive implementation satisfying them. We

model the adjustable reference type as its internal representation type, and have `aref_get` apply the function f to it. Adjusting reads simply ignore the adjustment and behave as normal reads.

```
Definition aref R T (f : R → T) :=  R.
Definition aref_val R T (f: R → T) (v : R) :=  v.
Definition aref_get R T (f: R → T) (r : aref f) :=  f r.
Definition aref_getu R A B (f : R → A → B) (upd : R → A → R * B)
       (PF: ∀x a, f (fst (upd x a)) = f x)
       (PF': ∀x a, snd (upd x a) = f x a) :=  aref_get f.
Definition aref_new R1 T1 (f1: R1 → T1) (r: aref f1)
    R2 T2 (f2 : R2 → T2) (g: ∀v, aref_get f1 r = f1 v → R2)
    (PF: ∀x pfx y pfy, f2 (g x pfx) = f2 (g y pfy)) :=
    g r eq_refl.
```

With this representation, it is straighforward to prove the associated axioms: our proof scripts are 'one-liners.'

2.2 Extraction to Efficient Imperative Code

Coq's extraction mechanism [7] generates OCaml code from the Coq definitions by erasing proofs and inserting suitable type casts to work around OCaml's type system. For types and operations that are specified as parameters, such as the `aref`, Coq allows us to specify their OCaml implementations.

There is also a way, using the `Extraction Implicit` directive, to remove further arguments (besides the propositional ones), if we know that a certain argument will not be used by the OCaml implementation. We use this feature to remove the unused observation function argument `f` from `aref_val` and `aref_getu`, as well as `f1` and `f2` from `aref_new`.

In OCaml, we implement adjustable references as a mutable reference cells storing the representation type. A normal read, `aref_get`, just reads the contents of the cell and applies the observation function, f, to it. An adjusting read, `aref_getu`, applies the update function, u, instead, and updates the reference cell as appropriate. Finally, `aref_new` simply creates a new reference cell as expected. Unfortunately, extraction cannot fully remove the second argument of `g` despite it being a proposition; so we call `g` with a dummy second argument. Below, we show our OCaml implementation with the OCaml types in comments.

```
type ('r,'t) aref = 'r ref
let aref_val x = ref x     (* 'r → ('r,'t) ref *)
let aref_get f r = f !r    (* ('r → 't) → ('r,'t) aref → 't *)
let aref_getu u r a =
   let (v, b) = u !r a in r := v; b
   (* ('r → 'a → 'r * 'b) → ('r, 'a → 'b) aref → 'a → 'b *)
let aref_new r g = ref (g !r ())
   (* ('r1,'t1) aref → ('r1 → unit → 'r2) → ('r2,'t2) aref *)
```

3 Memoization Using Adjustable References

A simple use of adjustable references is in the memoization optimisation. Given a function $f : A \to B$, we construct the function `memo` f which is extensionally equal to f, but which caches the results of previous f invocations, so that if `memo` f is called with the same argument again, the cached version is used. To implement the cache, we also require a decidable equality on A, as well as a hash function mapping elements of A to machine integers. We use the Coq **Section** mechanism to avoid repeating these assumptions for every definition.

```
Section Memo.
Variables (A B : Type) (f : A → B).
Variable eqA : ∀x y : A, { x = y } + { x ≠ y }.
Variable hash : A → int.
```

The cache is just an array of pairs (a, b) such that $f(a) = b$. In Coq,

```
Definition cache :=
  { c : Parray.t (option (A * B)) |
      ∀ x a b, Parray.get c x = Some (a, b) → b = f a }.
```

where we assume a module, `Parray`, implementing functional arrays.

The main program, `memo`, creates an adjustable reference cell holding an initially empty cache that represents the function f, and then returns a function that does an adjusting read from that reference cell. The main work of the adjusting read is performed by the `memo_upd` function, which reads the cache to determine if it contains an appropriate entry (a, b): if so, it returns b; if not, it calculates $f(a)$ and stores $(a, f(a))$ into the cache. In case of a hash collision, for simplicity, we simply overwrite the old colliding array entry.

We define these operations using the **Program** feature of Coq which allows us to write functions in a natural style and emits missing proof obligations as goals to be proved interactively at the end of the definition. In this example, we are basically asked to show that the initial and updated arrays are valid caches. For simplicity, we omit these easy proofs.

```
Program Definition memo_upd (c : cache) (a : A) : (cache * B) :=
  let h :=  hash a in
  match Parray.get c h with
    | None ⇒
        let b := f a in (Parray.set c h (Some (a, b)), b)
    | Some (a', b) ⇒
        if eqA a a' then (c, b) else
        let b := f a in (Parray.set c h (Some (a, b)), b)
  end. ⟨...⟩

Program Definition memo :=
  let r := aref_val (fun c : cache ⇒ f)
             (Parray.create (Int.repr 100) None) in
  aref_getu memo_upd _ _ r. ⟨...⟩
```

We can now easily prove that the memo function is equivalent to f: we unfold the definition of `memo` and rewrite using two adjustable reference axioms.

Lemma `memo_eq : memo = f.`
Proof. `by unfold memo; rewrite aref_getuE, aref_get_val.` **Qed.**

Finally, we close the Coq section, which will parametrize all the functions and lemmas declared within the section by the variables `A`, `B`, `f`, `eqA`, and `hash`.

End `Memo.`

4 Union-Find Path Compression

As a second example, we implement the path compression optimisation, which is crucial for achieving good performance in the union-find algorithm [12].

The union-find data structure describes a partition of a finite set, and supports two operations: (1) `find` returning the representative of an element (such that two elements are in the same partition iff they have the same representative), and (2) `union` that coalesces two partitions.

The data-structure is organised as an upward pointing forest so that elements of the same partition belong to the same tree. In this setting, `find` follows the parent-pointing edges from its argument until it reaches the root of its tree, which it returns as the representative, whereas `union` simply adds an edge from the root of the one partition to the root of the other.

To achieve practically constant (inverse Ackerman) time per operation, `find` 'compresses' the paths during look up. There are many ways of doing so, the simplest being to make all the nodes along the path from the input node to the root point directly to the root.

Below, we implement two functions that do the path lookup: `get_aux` (simply returning the root) and `find_aux` (also returning the updated path-compressed graph). When writing these functions in Coq, we have (as an orthogonal problem) to prove termination for the path lookups. For conciseness, however, we omit these proofs from the presentation.

```
Definition get_rel (a : Parray.t int) (x y : int) : Prop :=
  x ≠ y ∧ Parray.get a y = x.

Program Fixpoint get_aux a x (WF : Acc (get_rel a) x) :=
  let y := Parray.get a x in
  if y ≡ x then x else get_aux a y ⟨...⟩.

Program Fixpoint find_aux a x (WF : Acc (get_rel a) x) :=
  let y := Parray.get a x in
  if y ≡ x then (a, x)
  else let '(f, r) := find_aux a y ⟨...⟩ in
       (Parray.set f x r, r).
```

Our main data structure then consists of a rank array returning an approximate size of each partition (so that when `union` merges two partitions, it makes the smaller point to the larger one) and an adjustable reference to the parent-pointing array representing the union-find forest. Formally, we represent the latter as a refinement type, as we need to ensure that it has the same length as the rank array and actually represents a forest (so that path lookups terminate).

```
Definition closed_arr_cond (length: int) (a: Parray.t int) :=
  Parray.length a = length
  ∧ (∀ x, Int.ltu (Parray.get a x) (Parray.length a))
  ∧ well_founded (get_rel a).

Definition closed_arr length := { a | closed_arr_cond length a }.

Record t := { ranks : Parray.t int ;
              parr : aref (@get_closed (Parray.length ranks)) }.
```

With these definitions, it is now easy to implement the main union-find operations. The omitted proof obligations have to do with ensuring 'forestness' is maintained when compressing paths or adding edges, and the soundness of `find_aux` with respect to `get_aux`.

```
Program Definition create (size: int) : t :=
  {| ranks :=  Parray.create size Int.zero ;
     parr :=  aref_val _ (Parray.init size id) |}.

Definition find (uf: t) : int → int :=
  aref_getu (fun a x ⇒ @find_aux (proj1_sig a) x ⟨...⟩) ⟨...⟩⟨...⟩
    (parr uf).

Definition union (uf : t) (a b : int) : t :=
  let a' :=  find uf a in
  let b' :=  find uf b in
  if a' ≡ b' then uf
  else
    let ra :=  Parray.get (ranks uf) a' in
    let rb :=  Parray.get (ranks uf) b' in
    if Int.ltu ra rb then
      {| ranks :=  Parray.set (ranks uf) b' (Int.add ra Int.one) ;
        parr :=  aref_new (parr uf) (fun r PF ⇒
          existT _ (Parray.set (proj1_sig r) b' a') ⟨...⟩) ⟨...⟩|}
    else
      {| ranks :=  Parray.set (ranks uf) a' (Int.add rb Int.one) ;
        parr :=  aref_new (parr uf) (fun r PF ⇒
          existT _ (Parray.set (proj1_sig r) a' b') ⟨...⟩) ⟨...⟩|}.
```

In our Coq development, we proceed further to prove various properties about `create`, `find`, and `union` which together entail the correctness of our union-find implementation. We also use this union-find data structure to implement an efficient certified separation logic satisfiability checker.

5 Conclusion

This paper has presented adjustable references, a referentially transparent data type that enables imperative programming in a local and semantically unobservable fashion. We have seen how adjustable references can be used to implement and verify memoization and path compression, two important optimisations that cannot be programmed in a purely functional style.

One clear omission from this paper is a formal proof that the efficient imperative OCaml implementation of §2.2 is equivalent to the naive one of §2.1. Formally, we have to prove contextual equivalence in the context of a language with higher-order state, and a type and effect system including abstract, recursive and dependent types. To show that the two implementations are equivalent, one would have to extend the proof techniques for showing contextual equivalence, such as Kripke logical relations [10] or relation transition systems [6], to this setting. This task is by no means trivial, and is left as future work.

This work was largely inspired by a paper by Conchon and Filliâtre [5], who built persistent array and union-find implementations, whose performance is very close to that of the standard imperative implementations. In that paper, Conchon and Filliâtre used Coq to verify a monadic encoding of a slightly simplified form of those implementations against an axiomatisation of ML references. Adjustable references allows us to make a step further and program the exact path-compressing union-find algorithm directly in Coq.

It should be noted, however, that adjustable references are not a panacea. For example, they cannot directly be used to program the Conchon and Filliâtre persistent array [5]. The issue is that their implementation performs a sequence of updates that temporarily change the externally observable values of reference cells only to restore them at the end of the sequence. By definition, adjustable references do not permit such value-changing updates. To program such persistent data structures we need a more general primitive that can take a sequence of updates to multiple references and check that the entire sequence does not alter the observable value of any individual cell.

In essence, what we would like to have is an adjustable state monad, within which unrestricted read-write access to the internal values of adjustable references is allowed, together a primitive operation for converting internally stateful computations into pure computations:

$$\texttt{runST} : \forall c : \mathsf{AdjStateMonad}\ A.\ c \text{ is logically pure} \to A\,.$$

The somewhat informal condition that c is logically pure is supposed to check that (1) c's output is independent of the internal values of any reference cells it accessed, and (2) c's end-to-end behaviour does not change the external values of

any reference cells that were not newly created by its execution. Again, properly defining `runST` and formally justifying its soundness seems quite a challenging task, which we leave for future work.

Even though adjustable references as presented in this paper are clearly not applicable to every internally imperative, persistent data structure, we have identified and presented two examples that can easily be programmed with them. We hope that they will be equally useful in programming other similar persistent data structures directly inside interactive theorem provers.

Acknowledgements. I would like to thank Beta Ziliani and the anonymous ITP 2013 reviewers for their constructive feedback, which helped improve the presentation of the paper.

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. Aydemir, B.E., et al.: Mechanized Metatheory for the Masses: The PoplMark Challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
3. Bertot, Y.: A short presentation of Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 12–16. Springer, Heidelberg (2008)
4. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI 2011, pp. 234–245. ACM (2011)
5. Conchon, S., Filliâtre, J.-C.: A persistent union-find data structure. In: Russo, C.V., Dreyer, D. (eds.) ML 2007, pp. 37–46. ACM (2007)
6. Hur, C., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and Kripke logical relations. In: POPL 2012, pp. 59–72. ACM (2012)
7. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
8. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Functional Programming 18(5-6), 865–911 (2008)
9. Nanevski, A., Vafeiadis, V., Berdine, J.: Structuring the verification of heap-manipulating programs. In: POPL 2010, pp. 261–274. ACM (2010)
10. Pitts, A.M., Stark, I.D.B.: Operational Reasoning for Functions with Local State. In: Gordon, A.D., Pitts, A.M. (eds.) Higher Order Operational Techniques in Semantics, pp. 227–273. CUP (1998)
11. 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)
12. Tarjan, R.E., Van Leeuwen, J.: Worst-case analysis of set union algorithms. JACM 31(2), 245–281 (1984)
13. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)
14. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: ICFP 2004, pp. 175–188. ACM (2004)

Handcrafted Inversions Made Operational on Operational Semantics

Jean-François Monin[1,2] and Xiaomu Shi[1]

[1] Université de Grenoble 1 - VERIMAG
[2] CNRS - LIAMA

Abstract. When reasoning on formulas involving large-size inductively defined relations, such as the semantics of a real programming language, many steps require the inversion of a hypothesis. The built-in "inversion" tactic of Coq can then be used, but it suffers from severe controllability, maintenance and efficiency issues, which makes it unusable in practice in large applications.

To circumvent this issue, we propose a proof technique based on the combination of an antidiagonal argument and the impredicative encoding of inductive data-structures. We can then encode suitable helper tactics in LTac, yielding scripts which are much shorter (as well as corresponding proof terms) and, more importantly, much more robust against changes in version changes in the background software. This is illustrated on correctness proofs of non-trivial C programs according to the operational semantics of C defined in CompCert.

1 Introduction

The work described here is motivated by an experiment reported in [3,14], called SimSoc-Cert (a certified simulator of Systems on Chips) where we develop proofs of C programs using the operational semantics of a large subset of the C language as defined in the CompCert project [6]. An important characteristic of our framework is the large complexity of the specification, driving us to use powerful features such as higher-order functions, dependent types, modules, not only for convenience, but in order to keep the specification as readable and reusable as possible. Still, its size is rather large by force, since it includes the behavior of several commercial processors (currently: ARM and SH4). In such a framework, there is little hope for full automation. Proofs are performed by alternating clues given by the human user and tedious steps that are expected to be automated. Though all this is well-known, the situation can become very tricky when automated steps produce goals with many new variables and hypotheses in the environment. In an interactive setting, their names can be refered to later in the script. This issue cannot be overlooked, despite the lack of a nice theory on massive names management – up to our knowledge. And it actually occurs with SimSoc-Cert, because proofs rely heavily on *inversion* steps on hypotheses relating memory states of the program, according to a large inductive transition relation which is the heart of the operational semantics of C defined in CompCert.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 338–353, 2013.

In a few words, an inversion is a kind of forward reasoning step, which allows us to extract all useful information contained in a hypothesis. It is nothing but a case analysis on a carefully prepared goal (more detail to come in Section 2). The practical need for automating inversion has been identified many years ago and most proof assistants (Isabelle, Coq, Matita,...) provide an appropriate mechanism. The first implementations for Coq and LEGO are analyzed and explained in [5] for Coq and [7] for LEGO. Since then, the main tool available to the Coq user is a tactic called `inversion` which, basically performs a case analysis over a given hypothesis according to its specific arguments, removes absurd cases, introduces relevant premises in the environment and performs suitable substitutions in the whole goal. This tactic works remarkably well, though it fails in rare intricate cases, as reported in mailing lists (see also Section 3.5). An additional approach called BasicElim was proposed in [8]. It is implemented in Matita [13], for instance. BasicElim is available in Coq as well.

However, the price to pay for the generality of `inversion` and BasicElim is a high complexity of underlying proof-terms. Does it reflect an unnecessarily complex formalization of a (at first sight) rather simple idea? A practical consequence is that unpleasantly heavy proof terms can unexpectedly occur in functions defined in interactive mode. For developments which make an intensive use of inversion, such as SimSoc-Cert, the evaluation of scripts is painfully slowed down.

However, the abovementioned issue on name management turns out to be still much more important: hardly controlled names are introduced in the environment. This would not be an issue if we don't see them, e.g., if the generated goals can be automatically discharged. But this is hopeless when dealing with complex specifications, as in our case. In general, the sequel of the script refers to generated hypotheses. Typically, introduced hypotheses could be inverted again, and so on. This poses a very serious problem of robustness: updating the inductive relation or even minor modifications in another part of the development may result in a complete renaming inside a proof script, which has then to be debugged line by line. In the previous stage of our work reported in [14], we could perform a proof on a single instruction of the ARM processor. So in theory, everything was solved. However, the number of inversion steps was so large this proof could not survive the various updates of Coq and CompCert.

The available version of `inversion` where explicit names can be given in the script (`inversion... as`) is better for robustness, but too heavy for our needs: each inversion would require the introduction of many (often more than ten) additional names. BasicElim raises similar issues, though its behaviour is more regular.

In order to get scripts which are both robust and much shorter, we want to provide programmable inversion tactics, requiring only a few explicit names. To this effect, we propose a handcrafted approach to inversion. The initial idea for this inversion was exposed in [9] (and is recalled here in Section 3.2) but, in order to be general enough, it had to be revisited with inspiration coming from the impredicative encoding of inductive datatypes.

The concrete setting considered here is the Coq proof assistant, but the technique can be adapted to any proof assistant based on the Calculus of Inductive constructions or a similar type theory, such as LEGO or Matita.

The rest of the paper is organized as follows. Section 2 recalls the basics on inversion. Section 3 explains our technique for performing inversions. Section 4 contains a summary of the application to SimSoC-cert. We conclude in Section 5 with a comment on our achievements and some perspectives.

2 Inversion

Type-theoretic settings such as Coq [15,2,4] offer two elementary ways of constructing new objects: functions and inductive types[1]. For instance, even Peano natural numbers can be inductively characterized by the following two rules:

$$\frac{}{even\_i\ 0}\ E0 \qquad \frac{even\_i\ n}{even\_i\ (S\ (S\ n))}\ E2$$

Rules *E0* and *E2* serve as canonical justifications for *even_i*, they are called the *constructors* of the inductive definition.

Now, assume a hypothesis H claiming that *even_i (S (S (S x)))* for some natural number x. Then, by looking at the definition of *even_i*, we see that only *E2* could justify H, and we can conclude that *even_i (S x)*. Similarly, *even_i* 1 can be considered as an absurd hypothesis, since *(S 0)* matches neither 0 nor *(S (S n))*, none of the two possible canonical ways of proving *even_i*, namely *E0* and *E2* can be used. Such proof steps are called *inversions*, because they use justifications such as *E0* and *E2* in the opposite way, i.e., from their conclusion to their premises. Note that *even_i* 3, *even_i* 5, etc. do not immediately yield the contradiction by inversion. However, by iterating the first inversion step, we eventually get *even_i* 1 and then the desired result using a last inversion. This illustrates that inversion is closer to case analysis than to induction.

Indeed, as we will see below, inversion can be decomposed into elementary proof steps, where the key step is a primitive case analysis on the considered inductive object (the hypothesis H, in our previous example). However, this decomposition is very often far from trivial because, in the general case, rules may include several premises, premises and conclusions may have several arguments and some of these arguments can be shared. Still, inversion turns out to be extremely useful in practice. Well-known instances are related to programming languages, whose semantics is described using complex inductively defined relations.

Note that it may be worth considering a (recursive) *function* for defining a predicate, rather than an inductive relation. For instance, in Coq syntax, an alternative way to specify even numbers is as follows:

[1] Co-inductive types are available as well. However, this paper does not depend on issues related to finiteness of computations: what is said about inductive types holds as well for co-inductive types.

```
Fixpoint even_f (n: nat) : Prop :=
  match n with
  | O ⇒ True
  | 1 ⇒ False
  | S (S n) ⇒ even_f n
  end.
```

Here *True* denotes a trivially provable proposition, and *False* denotes an absurd proposition. Using *even_f* is much simpler in the previous situations: for instance, *even_f (S (S (S x)))* just *reduces* to *even_f (S x)* using computation. In other words, computation provides inversion for free. Therefore, one may wonder why we should bother with inductively defined relations. Two kinds of answers can be given.

One of them is that an inductive definition allows us to focus exactly on the relevant values whereas, with functional definitions, we have to deal with the full domain, which can be much bigger in general. In our example above, suppose that we want to prove a statement such as $\forall n, even\ n \Rightarrow P\ n$. We can always attempt an induction on n, but this strategy forces to reason on all numbers, including odd numbers. If *even* is the recursive function above *even_f*, there is no other option. However, using *even_i*, we have the additional opportunity to make an induction on (the shape of) $even\_i\ n$, without needing to bother about odd numbers.

Another issue is that it is not always convenient or even possible to provide a functional definition of a predicate. Whenever possible, an n-ary relation R on $A_1 \times \ldots A_n$, is advantageously modeled by a function from $A_1, \ldots A_{n-1}$ to A_n. But it requires R to be functional (deterministic) and moreover, in type-theoretical settings such as CIC, to be total. If the relation is non-deterministic, we still can try to define it by a function returning either *True* or *False*, as is the case for *even_f*; this essentially amounts to providing a decision procedure for the intended predicate[2]. This is not always possible and, even if we can find such an algorithm, it may be hindered by undesired encoding tricks, which will induce additional complications in proofs. Moreover, a requirement of formal methods expresses that high-level definitions and statements should be as clear as possible in order to be convincing. The inductive style is not always better than the functional style, but it is often enough the case so that we cannot ignore it. For technical reasons, it is sometimes worth considering a functional version and an inductive version of the same notion. Even if the functional version is much better at inversion-like proof steps, the two versions have to be proved equivalent and there, the need for inverting the inductive version almost inevitably shows up.

Inductive relations are commonly used for defining the operational semantics of programming languages, either in small-step or in big-step style [11]. Such semantics define transitions between states, language constructs and, very often,

[2] Note that a 1-ary relation P on A_1 is isomorphic to a binary relation on $\mathbf{1} \times A_1$, where $\mathbf{1}$ is a type with exactly one inhabitant. If P holds for at least two values on A_1, it can be clearly considered as a non-deterministic function from $\mathbf{1}$ to A_1.

additional arguments such as input/output events. A tutorial example of a toy (but Turing-complete) language formally defined in Coq along these lines is given in [12] and routinely used as a teaching support in many universities. The Compcert project [6] is of course a much more involved example.

3 A Handcrafted Inversion

As noticed above, the heart of inversion is a suitable pattern matching on the hypothesis to be analyzed. With dependent types, it is possible for different branches to return a result whose type depends on the constructor. We make a systematic use of this feature: our key ingredient is a diagonalization function *diag*, which will be used for specifying the type returned on each branch. The exact shape of *diag* range from very simple to somewhat elaborated according to the goal at hand.

We recall the basics on dependent pattern matching, then we successively consider three situations, corresponding to increasingly complex variants of *diag*. In the two first situations, we consider inductive predicates with exactly one argument, for simplicity. The first situation is when all cases are absurd. The second is when a case is successful (or several cases) and we need to extract the information contained in successful cases, making new hypotheses in the environment. Then we show how to deal with additional arguments, so that constraints coming from the conclusion have to be propagated on the new hypotheses. Finally, we consider more elaborate dependent types and show how our technique works on a case where `inversion` fails.

3.1 Dependent Pattern Matching

To start with, let us take again the example of even numbers. Here is the corresponding Coq inductive definition.

```
Inductive even_i : nat → Prop :=
  | E0: even_i 0
  | E2: ∀ n, even_i n → even_i (S (S n)).
```

We see that each rule is given by a constructor in a dependent data type – also called an inductive predicate or relation because its sort is `Prop`. Therefore, the elementary way to decompose an object of type $even\_i\ n$ is to use dependent pattern matching. This is already done by primitive tactics of Coq such as `case` and `destruct`, which turn out to be powerful enough in many situations, when a condition is satisfied: the conclusion of the current goal fits all arguments of the hypothesis to be analyzed by pattern matching.

Let us first illustrate dependent pattern matching on even numbers. Consider a proof PE of type $even\_i\ n$ for some natural number n. For each possible constructor, *E0* or *E2*, we provide a proof term, respectively t_{E0} and t_{E2}. As usual, this term may depend on the arguments of the corresponding constructor, none for *E0* and, say x and ex for *E2*. More importantly for us, t_{E0} and t_{E2}

may have different *types*: the type $P\ n$ of the whole expression depends on n; in the first branch, the type of t_{E0} is $P\ 0$ and in the second branch, the type of t_{E2} is $P\ (S\ (S\ x))$. Therefore, the syntax of the `match` construct contains a `return` clause with the expected type of the result $P\ n$ as an argument; moreover, there is also an `in` clause for the type of PE which binds n:

```
match PE in even_i n return P n with
  | E0 ⇒ t_E0
  | E2 e ex ⇒ t_E2
end
```

Most of the time, Coq users do not need to go to this level of detail: in interactive proof mode, if n and $P\ n$ are clear from the context, `case` PE will do the job. More precisely, if we have an hypothesis H of type $even\_i\ n$ and a desired conclusion of type $P\ n$, `case` H will construct a proof term having the previous shape and answer with two new subgoals: one for $P\ 0$ and one for $P\ (S\ (S\ x))$, with $even\_i\ x$ as an additional assumption.

As a last remark, let us recall that an inductive type may have two kinds of arguments. We don't care about arguments which are "fixed" for all constructors: they are not even considered in pattern matching. In Coq they are called *parameters*. The other arguments are called *indexes*. For example, $even\_i$ has one index and no parameter.

3.2 Auxiliary Diagonalization Function

More work is needed precisely when there is no obvious relationship between the conclusion and the hypothesis to be analyzed. This happens in particular when H is absurd: the goal should be discharged whatever is its conclusion. This situation is covered as follows: the conclusion is converted to an expression $diag\ V$, where V is a value coming from H and $diag$ a suitable diagonal function, such that the dependent case analysis on H provides only trivial subgoals. For example, assume that we want to conclude $4 = 7$ from the hypothesis $H : even\_i\ 1$. Our diagonal function is then defined as follows.

$$diag\ x := \texttt{match}\ x\ \texttt{with}\ 1 \Rightarrow 4 = 7 \mid \_ \Rightarrow True\ \texttt{end}$$

Then the conclusion is converted to $diag\ 1$, and the case analysis on H automatically provides two subgoals $diag\ 0$ and $diag\ (S\ (S\ y))$ for an arbitrary even natural number y. Each of these goals reduce to $True$, and we are done. The proof term behind this reasoning is very short (I is the standard proof of $True$):

$$\texttt{match}\ H\ \texttt{in}\ even\_i\ n\ \texttt{return}\ diag\ n\ \texttt{with}\ E0 \Rightarrow I \mid E2\ \_\ \_ \Rightarrow I\ \texttt{end}$$

Such functions were already introduced in [9], but they work well only for handling absurd hypotheses. For instance, the examples presented below are out of reach of [9]. In order to explain how to extract information from satisfiable hypotheses, we start with an obvious generalization of the previous function for inverting absurd hypotheses.

3.3 Handling Successful Cases

A first easy improvement makes *diag* independent from the conclusion. To this effect, we replace it with $(\forall X, X)$ in the first branch of *diag*. In our previous example, this yields

```
diag x := match x with 1 ⇒ ∀ (X: Prop), X | _ ⇒ True end
```

Then the previous proof term (`match` H `in` *even_i n* `return` *diag n* `with` 1 ...) has the type $\forall X, X$ and then can be successfully applied to any current conclusion. Alternatively, we can define a general function as follows:

```
Definition pr_1 {n} (en: even_i n) :=
  let diag x := match x with 1 ⇒ ∀ (X: Prop), X | _ ⇒ True end in
  match en in even_i n return diag n with E0 ⇒ I | _ ⇒ I end.
```

Next consider the following theorem:

$\forall\ n\ m,\ even\_i\ n \to even\_i\ (n{+}m) \to even\_i\ m.$

The proof is by induction on *even_i n*. In the inductive step, we have to prove *even_i m* from the induction hypothesis $even\_i\,(n+m) \to even\_i\,m$ and a new hypothesis $H : even\_i\,(S\,(S\,(n+m)))$. Intuitively, we want to invert H in order to push $even\_i\,(n+m)$ in the environment. We can then adapt *pr_1* as follows:

```
Definition premises_E2 {n} (en: even_i n) :=
  let diag x :=
    match x with
      | S (S y) ⇒ ∀ (X: Prop), (even_i y → X) → X
      | _ ⇒ True
    end in
  match en in even_i n return diag n with
     | E2 p e ⇒ fun X k ⇒ k e
     | _ ⇒ I
  end.
```

Then, applying *premises_E2* to H yields a function in continuation passing style. Its type parameter X is automatically identified to the conclusion *even_i m*, while y is bound to $n+m$, so that we get a new goal $even\_i(n+m) \to even\_i\,m$. That is, we have exactly the expected inversion. Functions such as *pr_1* and *premises_E2* can be seen as inversion lemmas, but note that their type is the dependent type expressed by their own *diag*.

More generally, let us then invert an hypothesis H having the type $A\,\mathcal{P}$ where $A(u)$ is an inductive type with index $u : U$ and $\mathcal{P} : U$ is an expression made of constructors in the type U. Given a constructor of type $\forall \mathbf{p}, A\ p$, where $\mathbf{p}$ is a telescope we proceed similarly: the `match` of *diag* has a first branch filtering $\mathcal{P}$ and returning $\forall X : Prop, (\forall \mathbf{p}, X) \to X$. If n constructors are possible for $A\,\mathcal{P}$, say respectively $C_1 : \forall \mathbf{p_1}, A\,\mathcal{P}$, ..., and $C_n : \forall \mathbf{p_n}, A\,\mathcal{P}$, the inverting lemma corresponding to $A\,\mathcal{P}$ will be:

```
Definition premises_Ap {u} (a: A u) :=
  let diag x :=
    match x with
      | P ⇒ ∀ (X: Prop), (∀ p1, X) → ... (∀ pn, X) → X
```

```
        | _ ⇒ True
      end in
    match a in A p return diag p with
      | C1 e1  ⇒ fun X k1... kn ⇒ k1 e1
  ⋮
      | Cn en  ⇒ fun X k1... kn ⇒ kn en
      | _ ⇒ I
    end.
```

Remark the close relationship with the impredicative encoding of data-types in system F.

3.4 Dealing with Constrained Arguments

The next stage to be considered is the case of an inductive type with more than one index. This raises new issues, because additional identities between arguments of the premises or the conclusion of a constructor may occur. This happens routinely in the inductive definitions for the operational semantics of C provided by CompCert. In order to explain the problems and how to deal with them in our framework, we introduce a toy language, together with an inductively defined evaluation rule *eval* having two indexes: the first one is the input type *tm*, *tm_const* and *tm_plus* are the expected cases in pattern matching; the second index is an output of type *val*, which is either nat or bool.

```
Inductive tm : Type :=
  | tm_const : nat → tm
  | tm_plus : tm → tm → tm.
Inductive val : Type :=
  | nval : nat → val
  | bval : bool → val.
Inductive eval : tm → val → Prop :=
  | E_Const : ∀ n,
      eval (tm_const n) (nval n)
  | E_Plus : ∀ t1 t2 n1 n2,
      eval t1 (nval n1) → eval t2 (nval n2) →
      eval (tm_plus t1 t2) (nval (plus n1 n2)).
```

In constructor E_Plus, the two premises share the variables $t1, t2, n1, n2$ with the conclusion. If we use the last solution with continuation passing style, as it is presented above, we are able to keep the premises but the relationship between the output values as specified in the inductive definition will be lost in the generated subgoal. This issue is handled using an additional argument to X corresponding to the second index of the inductive relation. The function for extracting the premises of E_Plus is:

```
Definition pr_plus_1 {t} {v} (e: eval t v) :=
  let diag t v :=
  match t with
    | tm_plus t1 t2 ⇒ ∀ (X:val → Prop),
```

```
      (∀ n1 n2, eval t1 (nval n1) → eval t2 (nval n2) → X (nval (plus n1 n2)))
      → X v
  | _ ⇒ True
end in match e in (eval t v) return diag t v with
      | E_Plus _ _ n1 n2 H1 H2 ⇒ (fun X k ⇒ k n1 n2 H1 H2)
      | _ ⇒ I
    end.
```

Now, consider the following examples.

```
Lemma ex1: ∀ v, eval (tm_plus (tm_const 1) (tm_const 0)) v → v = nval 1.
Lemma ex2: ∀ n, eval (tm_plus (tm_const 1) (tm_const 0)) (nval n) → n = 1.
```

In *ex1*, by applying pr_plus_1, v will be equated to $nval\ (plus\ n1\ n2)$ according to the rule specified by E_plus. In *ex2*, we need to analyze at the same time the two arguments of *eval*. The corresponding premises are extracted using a function $pr\_plus\_1\_2$ having the same body as pr_plus_1, but whose type is:

```
match t, v with
  | tm_plus t1 t2, nval n ⇒ ∀ (X:nat → Prop),
    (∀ n1 n2, eval t1 (nval n1) → eval t2 (nval n2) → X (plus n1 n2)) → X n
  | _, _ ⇒ True
end.
```

A similar situation happens with E_Const in the two previous examples.

Defining an inverting function for each constructor is most convenient for debugging. However the method is flexible and several such functions can be merged. In particular, an elegant alternative[3] is to provide a unique inverting function managing all cases of the argument(s) under focus. For instance, an exhaustive inverting function $pr\_eval\_1\_2$ suitable for *ex2* has the type:

```
match t, v with
  | tm_const c, nval n ⇒ ∀ (X:nat → Prop), X c → X n
  | tm_plus t1 t2, nval n ⇒ ∀ (X:nat → Prop),
    (∀ n1 n2, eval t1 (nval n1) → eval t2 (nval n2) → X (plus n1 n2)) → X n
  | _, _ ⇒ ∀ X:Prop, X
end.
```

Full definitions as well as additional examples can be found on-line [10].

3.5 Beating `inversion`

Let us consider now a predicate defined on a dependent type. We take intervals $[1...n]$, formalized as t in the standard library `Fin`, then we restrict them to have an odd length.

```
Inductive t : nat → Set :=
    | F1 : ∀ {n}, t (S n)
    | FS : ∀ {n}, t n → t (S n).

Inductive odd : ∀ n : nat, t n → Prop :=
```

[3] We want to thank the anonymous reviewer who offered this remark.

```
  | odd_1 : ∀ n, odd (S n) F1
  | odd_SS : ∀ n i, odd n i → odd _ (FS (FS i)).
```

Finding the premises for the second constructor is a function similar to the one provided for $E2$ above:

```
Definition premises_odd_SS {n} {i: t n} (of: odd n i) :=
  let diag n i :=
    match i with
      | FS _ (FS _ y) ⇒ ∀ (X: Prop), (odd _ y → X) → X
      | _ ⇒ True
    end in
  match of in odd n i return diag n i with
      | odd_SS n i o ⇒ fun X k ⇒ k o
      | _ ⇒ I
  end.
```

In particular we can easily prove:

```
Lemma odd_SS_inv: ∀ n i, odd _ (FS (FS i)) → odd n i.
Proof. intros n i o. apply (premises_odd_SS o). trivial. Qed.
```

Standard `inversion` happens to fail here. Note that BasicElim may work (we actually could not succeed) but would need an additional axiom related to John Major equality.

4 Application to SimSoC-Cert

SimSoC-Cert [3,14] aims at certifying the simulator SimSoC, which is a complex hardware simulator written in C and C++. SimSoC is able to simulate various architectures including ARM and SH4 and is efficient enough to run Linux on them at a realistic speed. The main objective of SimSoC is to help designers of embedded systems: a large part of the design can be performed on software, which is much more convenient, flexible and less expensive than with real specific hardware components. However, this only makes sense if the simulator is actually faithful to the real hardware. Therefore we engaged in an effort to provide a formal certification of sensitive parts of SimSoC. More precisely, we consider the Instruction Set Simulator (ISS) for the ARM, which is at the heart of SimSoC. This ISS is called Simlight.

To this effect, first we defined a formal model in Coq of the ARM architecture, as defined in the reference manual [1]. Our second input is the operational semantics of the ISS encoded in C. This program is actually written in a large enough subset of C called Compcert-C, which is fully formalized in Coq [6].

We can then compare the behavior of the ISS encoded in C with the expected reference model directly defined in Coq. To this effect, a projection between the Coq model of the memory state of Simlight to the states in the reference model is defined. Then, correctness statements express that from a C memory m_1 corresponding to an abstract state s_1, performing the function claimed to represent a given instruction $\mathcal{I}$ in Simlight will result in a C memory m_2 which

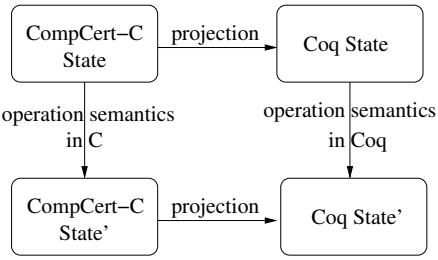


Fig. 1. Correctness of the simulation of an ARM operation

actually corresponds to the abstract state s_2 obtained by running the Coq model of $\mathcal{I}$. This can be put under the form of a commutative diagram as schematized in Fig. 1.

The operational semantics of C defining the evaluation is used everywhere in the proof: it provides the decomposition of the vertical arrow on the left column of Fig. 1 and drives the proof accordingly. We use the big step semantics, which is defined in CompCert by 5 mutually inductive transition relations. The largest inductive type for the evaluation of C expressions is *eval_expr*. It has 17 constructors, one for each CompCert C expression such as assignment, binary operation, dereference, etc.

In a typical proof step, we start from a goal containing a conclusion stating that a C memory state m_n and an ARM state st_n in the reference model are related by our projection, a hypothesis R_0 stating a similar relation between a C memory state m_0 and an ARM state st_0, and additional hypotheses $He_1, He_2, \ldots, He_n$. relating pairs of successive C memory states (m_0, m_1), $(m_1, m_2), \ldots, (m_{n-1}, m_n)$ respectively with (ASTs for) C expressions e_1, $e_2, \ldots$, e_n, according to the relevant transition relation provided by CompCert. The general strategy is to propagate information from m_0 to m_1 using R_0 and He_1, then so on until m_n. To this effect we invert He_1, He_2, etc. However, according to the structure of e_1, inverting He_1 generates intermediate memory states and corresponding hypotheses that have to be inverted before going to He_2, unless e_1 is a base case. And sometimes, other kind of reasoning steps are needed, e.g., lemmas on the reference model of ARM.

For illustration, the following code shows a small excerpt from an old proof script in SimSoC-Cert using `inversion`. It corresponds to one line taken in an instruction called ADC (add with carry). It sets the CPSR (Current Program Status Register) with the value of SPSR (Saved Program Status Register). Lemma *same_cp_SR* states that the C memory state of the simulator and the corresponding formal representation of ARM processor state evolve consistently during this assignment. The pseudo-code from the ARM reference manual is just $CPSR = SPSR$. The corresponding C code is represented by the identifier *cp_SR* in the statement of the lemma.

```
Lemma same_cp_SR :
  ∀ e m l b s t m' v em,
    proc_state_related proc m e (Ok tt (mk_semstate l b s)) →
    eval_expression (Genv.globalenv prog_adc) e m cp_SR t m' v →
    proc_state_related proc m' e
      (Ok tt (mk_semstate l b
      (Arm6_State.set_cpsr s (Arm6_State.spsr s em)))).
```

After a couple of introductions and other administrative steps, we get the following goal, where cp_SR is unfolded in hypothesis H.

```
  ...
  l' : local
  b' : bool
  a' : expr
  H : eval_expr (Genv.globalenv prog_adc) e m RV
        (Ecall (Evalof (Evar copy_StatusRegister T14) T14)
          (Econs
            (Eaddrof
              (Efield (Ederef (Evalof (Evar proc T3) T3) T6)
                adc_compcert.cpsr T7) T8)
            (Econs
              (Ecall (Evalof (Evar spsr T15) T15)
                (Econs (Evalof (Evar proc T3) T3) Enil) T8) Enil))
          T12) t m' a'
  ============================
   proc_state_related m' e st'
```

Then we have to invert H and similar generated hypotheses until all constructors used in it type are exhausted. Here 18 consecutive inversions are needed. Using `inv`, which performs standard `inversion`, clearing the inverted hypothesis and rewriting of all auxiliary equations, the sequel of the script started as follows.

```
inv H. inv H4. inv H9. inv H5. inv H4. inv H5.
inv H15. inv H4. inv H5. inv H14. inv H4. inv H3.
inv H15. inv H5. inv H4. inv H5. inv H21. inv H13.
```

The names used there ($H4$, $H9$, etc.) are not under our control. The program for simulating an ARM instruction usually contains expression more complex than in the example given here. And unfortunately there is no clear way to share parts of the proofs involved since the corresponding programs are rather specific, at least for instructions belonging to different categories.

The drawbacks of the standard tactic `inversion` presented in the introduction show up immediatly. A first clue is the response time of Coq when inverting hypotheses H_i. Compiling the proof script corresponding to one instruction took more than a minute. About the naming issue, the constructors we face have up to 19 variables and 6 premises, yielding 25 names to provide. We could try to automate this naming using an ad-hoc wrapper around `inversion`, but things are complicated by the fact that this inversion program inserts additional hypotheses putting equational constraints between variables of the inverted constructor.

There are different ways to state and to place such constraints, and different releases of Coq may make different choices. The BasicElim approach introduces equations as well but from on our experiments, generated goals are much more regular than with `inversion`. In contrast, our approach does not suffer from such interferences, so we are anyway in a better position.

First, we define the diagonal-based function for each constructor of *eval_expr*, following the lines given in the previous section. For example, the evaluation of a field is defined in CompCert by the following rule.

```
Inductive eval_expr :
  env → mem → kind → expr → trace → mem → expr → Prop :=
  ...
  | eval_field: ∀ e m a t m' a' f ty,
      eval_expr e m RV a t m' a' →
      eval_expr e m LV (Efield a f ty) t m' (Efield a' f ty)
```

We then define (observe that 2 variables and 1 hypothesis will be generated):

```
Definition inv_field {g} {e} {m} {ex} {t} {m'} {ex'}
  (ee:eval_expr g e m LV ex t m' ex') :=
  let diag e ex ex' m m' :=
    match ex with
      | Efield a b c ⇒
        ∀(X:expr→Prop),
          (∀ t a', eval_expr g e m RV a t m' a' → X (Efield a' b c)) → X ex'
      | _ ⇒ True
    end in
    match ee in (eval_expr _ e m _ ex _ m' ex') return diag e ex ex' m m' with
      | eval_field _ _ _ t _ a' _ _ H1 ⇒ fun X k ⇒ k t a' H1
      | _ ⇒ I
    end.
```

Next we introduce a high-level tactic for each inductive type, gathering all the functions defined for its constructors. For example, *eval_expr* contains:

```
Ltac inv_eval_expr m m' :=
  ...
  let t1_:=fresh "t" in
  let v1_:=fresh "v" in
  let ev_ex1 := fresh "ev_ex" in
  ...
  match goal with
  ...
    | [ee: eval_expr ?ge ?e m LV (Efield ?a ?f ?ty) ?t m' ?a' ⊢ ?cl] ⇒
      apply (inv_field ee); clear ee; intros t1_ a1_ ev_ex1; intros;
      inv_eval_expr m m'
```

This tactic has two arguments m and m', corresponding to C memory states. The first `intros` introduces the 3 generated components with names respectively

prefixed by `t`, `v` and `ev_ex`. The second `intros` is related to previously reverted hypotheses, their names are correctly managed by Coq. Alltogether, such a tactic will:

1. Automatically find the hypothesis matching the arguments to be inverted;
2. Repeatedly perform our hand-crafted inversions for type *eval_expr* until all constraints between two memory states m and m' are derived;
3. Give meaningful names to the derived constraints;
4. Update all other related hypotheses according to the new variable names or values;
5. Clean up useless variables and hypotheses.

For example the 18 `inv` in the example above are solved in one step using *inv_eval_expr* m m', Note that the names are not explicitly given in the script, which would be cumbersome, but generated in our tactic.

Coq version changes had no impact on our scripts. Unexpectedly, changes in CompCert C semantics between versions 1.9 and 1.11 had no impact as well on proof scripts using our inversion. Of course, we still had to update the definition of diagonal functions.

Comparing development times provides additional hints. In our first try, using built-in inversion, more than two months were spent (by one person) on the development of the correctness proof of instruction ADC. Much time was actually wasted at maintaining the proofs since, as mentioned, a little change resulted in a complete revision of proof scripts. We then designed the inversion technique presented here. With the new approach, proofs for 4 other simple instructions could be finished in only one week, taking of course advantage of the previous experience with ADC. The high-level tactic described above required less than 2 weeks.

Finally, let us compare the efficiency of Coq built-in inversions (`inversion`, `derive inversion` which can generate an inversion principle once for all, and BasicElim [8]) with our inversion. We apply the four methods to the same examples, the lemma *cp_SR* and a single inversion on type *eval_expr* from CompCert C semantics. The first row is about the whole expression given in the example above. The other rows are inversions of specific expressions: *Ecall* is the CompCert-C expression of function calls, *Evalof* is to get the value of the specified location, *Eval* is to express constant, and *Evar* is to express variables. We can observe a gain of about 4 to 5 times. And generated object files are 5 times smaller.

Table 1. Time costs (in seconds)

	standard `inversion`	`derive inversion`	BasicElim	our inversion
Full example	1.628	0.976	1.428	0.312
Ecall	0.132	0.076	0.112	0.028
Evalof	0.132	0.072	0.092	0.020
Evar	0.128	0.064	0.084	0.024
Eaddrof	0.140	0.076	0.104	0.020

Table 2. Size of compilation results (in KBytes)

	standard `inversion`	`derive inversion`	BasicElim	our inversion
Full example	191	460	171	37

5 Conclusion

We see no reason why the technique developed above for performing inversions could not be automated and implemented in Coq or in proof assistants based on a similar calculus. One good motivation for that would be to get terms which are much smaller, easier to typecheck, than with the currently available inversion tactics. This can be very useful when interactively defining functions on dependent types, for instance.

But we want to insist first on a much more important feature of our approach, according to our experience with SimSoC-Cert: its impact on goals during *interactive* proof development is *actually controllable*. We think that having much shorter underlying functions is helpful in this respect: they are short enough to be written by hand, providing an exact view on what is to be generated. We claim that this feature is especially relevant to applications which make an intensive use of inversion steps: in this situation, partial automation obtained by programming small controllable building blocks turns out to be effective, whereas automation tends to generate a response of the proof assistant which is not completely predictible. This may not harm too much if the generated goals can be fully discharged without further interaction, but this is not the general case. In particular, this hope is vain when we deal with complex properties, as in our application. A better alternative would be to automatically generate auxiliary definitions such as *inv_field*. However, we consider that our technique is already useful and worth to be offered.

In contrast to available techniques [5,8] we argue against the use of auxiliary equations or disequations: the latter are better to be cleaned, in order to avoid clumsy additional hypotheses, which hamper the management of proof scripts; however, it is not that simple to do. The brute use of a tactic which performs all possible rewriting steps, then cleans equalities avalaible in the goal, for instance, is not satisfactory because some equalities already introduced by the user on purpose could then disappear. Therefore, a special machinery is needed in order to trace equalities coming from the inversion step under consideration (e.g., the use of *block* in BasicElim). Our use of CPS encoding of Leibniz equality, on the other hand, completely avoids this issue.

Our method was experimented on large proofs relying on big inductive relations independently defined in the Compcert project.

The current development can be found on-line [10], as well as examples given in Section 3.

Our group recently started another project dedicated to a certifying compiler from a high-level component-based language dedicated to embedded systems

(BIP), with CompCert C as its target. We expect the work presented here and our high-level tactics to be reused there.

Let us mention another possible application of the technique. Inversion is sometimes needed to write a function whose properties will be established later (as opposed to providing a monolithic and exhaustive Hoare-style specification and along with a VC generator such as Program). In this context simply using the proof engine and the `inversion` tactic tends to generate unmanageably large terms. We expect our technique to be very helpful in such situations.

References

1. ARM. ARM Architecture Reference Manual DDI 0100I. ARM (2005)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
3. Blanqui, F., Helmstetter, C., Joloboff, V., Monin, J.-F., Shi, X.: Designing a CPU model: from a pseudo-formal document to fast code. In: Proceedings of the 3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Heraklion, Greece (January 2011)
4. Chlipala, A.: Certified Programming with Dependent Types (2012), `http://adam.chlipala.net/cpdt`
5. Cornes, C., Terrasse, D.: Automating inversion of inductive predicates in coq. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 85–104. Springer, Heidelberg (1996)
6. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
7. 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)
8. 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)
9. Monin, J.-F.: Proof Trick: Small Inversions. In: Bertot, Y. (ed.) Second Coq Workshop, Royaume-Uni Edinburgh. Yves Bertot (July 2010)
10. Monin, J.-F., Shi, X.: Coq Examples for Handcrafted Inversions (2013), `http://www-verimag.imag.fr/~monin/Proof/hc_inversion/`
11. Nielson, H.R., Nielson, F.: Semantics with applications: A formal introduction. John Wiley & Sons, Inc., New York (1992)
12. Pierce, B.C., Casinghino, C., Greenberg, M.: Software Foundations (2009), `http://www.cis.upenn.edu/~bcpierce/sf`
13. Ricciotti, W.: Theoretical and Implementation Aspects in the Mechanization of the Metatheory of Programming Languages. PhD thesis, Università di Bologna (2011)
14. Shi, X., Monin, J.-F., Tuong, F., Blanqui, F.: First Steps Towards the Certification of an ARM Simulator Using Compcert. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 346–361. Springer, Heidelberg (2011)
15. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), `http://coq.inria.fr`

Circular Coinduction in Coq Using Bisimulation-Up-To Techniques*

Jörg Endrullis[1], Dimitri Hendriks[1], and Martin Bodin[2]

[1] VU University Amsterdam, Department of Computer Science
[2] INRIA Rennes & ENS Lyon
{j.endrullis,r.d.a.hendriks}@vu.nl, martin.bodin@inria.fr

Abstract. We investigate methods for proving equality of infinite objects using circular coinduction, a combination of coinduction with term rewriting, in the Coq proof assistant. In order to ensure productivity, Coq requires the corecursive construction of infinite objects to be guarded. However, guardedness forms a severe confinement for defining infinite objects, and this includes coinductive proof terms. In particular, circular coinduction is troublesome in Coq, since rewriting usually obstructs guardedness. Typically, applications of transitivity are in between the guard and the coinduction hypothesis. Other problems concern the use of lemmas, and rewriting under causal contexts. We show that the method of bisimulation-up-to allows for an elegant rendering of circular coinduction, and we use this to overcome the troubles with guardedness.

1 Introduction

As any construction of infinite objects, constructive bisimilarity proofs have to be *productive*. That is, it has to be guaranteed that the proof term has an infinite constructor normal form with respect to the lazy evaluation of the calculus at hand [5]. One way of ensuring productivity is by *guarded corecursion* [5,11]. Guardedness is a simple syntactic criterion implemented in proof assistants based on type theory like Coq [4] and Agda [2]. A corecursive definition is guarded if every corecursive call is guarded by at least one constructor of the coinductive type we are building a term in, and only by such constructors. Then, in the infinite process of unfolding a guarded definition, evermore building blocks of the infinite structure are produced, yielding in the limit a term consisting of constructors only.

Guardedness can be easily checked and it is readily seen why guarded corecursion implies productivity. On the other hand, guardedness is notorious for confining the programmer to a restricted set of tools for defining coinductive objects. Already the most simple examples of productive definitions fail to be guarded. Coquand [5] considers the following corecursive definition

$$\mathsf{nats} = 0 :: \mathsf{map}\ (\lambda n.\ n+1)\ \mathsf{nats} \qquad \mathsf{map}\ f\ (x :: s) = f\ x :: \mathsf{map}\ f s \tag{1}$$

* This research has been funded by the Netherlands Organization for Scientific Research (NWO) under grant numbers 639.021.020 and 612.000.934.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 354–369, 2013.

of the sequence of natural numbers $\mathsf{nats} = 0::1::2::\ldots$; where $::$ is the constructor of the coinductive type of infinite sequences, or streams as we call them. The definition of nats is clearly productive, yet it is not guarded, for the recursive call is argument of the map function.

A similar problem occurs for definitions of morphic sequences (see Section 2), like the following definition of the Thue-Morse sequence $\mathsf{M} = 0::1::1::0::1::0::0::1::1::0::0::1::0::1::1::0::\ldots$,

$$\mathsf{M} = 0 :: \mathsf{tail}\ (\mathsf{h}\ \mathsf{M}) \qquad \mathsf{h}\ (0 :: s) = 0 :: 1 :: \mathsf{h}\ s \qquad \mathsf{h}\ (1 :: s) = 1 :: 0 :: \mathsf{h}\ s \tag{2}$$

where $\mathsf{tail}\ (x :: s) = s$. The corecursive call of M is not a direct argument of $::$, and so Coq rejects this productive definition.

As indicated by Coquand [5], the problem of guardedness in (1), the definition of nats, can be overcome by the alternative definition

$$\mathsf{nats} = \mathsf{nats\_from}\ 0 \qquad\qquad \mathsf{nats\_from}\ n = n :: \mathsf{nats\_from}\ (n+1)$$

The 'computation' is now embedded in the argument of the corecursion, and no longer obstructs the guarding constructor $::$. In Section 2 we give a similar solution for (2), by generalizing the corecursive construction to carry a nonempty list as argument. We show that every morphic sequence can be defined in Coq.

The main objective of this paper is to enable the use of circular coinductive rewriting [12] in Coq. In type theories, where proofs are first-class citizens, the problem of guardedness also occurs in proving coinductive statements. In particular, equational reasoning and rewriting on terms of a coinductive type may very well destroy guardedness. Let us consider an example where we want to show that two stream terms are bisimilar. Bisimilarity as a relation between streams can be defined coinductively as follows (where head is defined by $\mathsf{head}\ (x::s) = x$):

$$\frac{\mathsf{head}\ s = \mathsf{head}\ t \qquad \mathsf{tail}\ s \sim \mathsf{tail}\ t}{s \sim t}\ \sim_{\mathsf{intro}}$$

This means that $\sim$ is the greatest bisimulation (a bisimulation is a relation R such that for all stream terms related by R the heads are equal and the tails are again related by R). Let s and t be closed stream terms (i.e., containing no variables, only constants from a given signature). A proof of $s \sim t$ evaluates, in the limit, to an infinite constructor normal form $\sim_{\mathsf{intro}}\ d_0\ (\sim_{\mathsf{intro}}\ d_1\ (\sim_{\mathsf{intro}}\ d_2\ \cdots))$ where d_i is a proof of the equality of the i-th elements of s and of t, that is, $d_i : \mathsf{head}\ (\mathsf{tail}^i\ s) = \mathsf{head}\ (\mathsf{tail}^i\ t)$ for all $i \in \mathbb{N}$.[1]

Suppose we want to prove that alt is bisimilar to $\mathsf{g}\ \mathsf{alt}$, given the assumptions[2]:

$$\mathsf{alt} \sim 0 :: 1 :: \mathsf{alt} \qquad \mathsf{g}\ (0 :: s) \sim 0 :: 1 :: \mathsf{g}\ s \qquad \mathsf{g}\ (1 :: s) \sim \mathsf{g}\ s \tag{3}$$

[1] Throughout the paper, we use $=$ to denote Coq's equality, defined as the $\subseteq$-least reflexive relation, equivalent to Leibniz equality. We note that $=$ includes convertibility induced by Coq's native evaluation.

[2] The flexibility to use assumptions, in addition to definitions, is essential to allow for application of lemmas, and for unguarded or partial specifications of objects and functions. E.g., the second equation for g is not guarded. In fact, g is productive only for streams that contain infinitely many 0s.

Our coinductive proof that alt is a fixed point of g has the following shape:

$$\dfrac{d_0 \qquad \dfrac{d_1 \qquad d'' : \mathsf{tail}^2\ \mathsf{alt} \sim \mathsf{tail}^2\ (\mathsf{g}\ \mathsf{alt})}{\mathsf{tail}\ \mathsf{alt} \sim \mathsf{tail}\ (\mathsf{g}\ \mathsf{alt})}\ {\sim_{\mathsf{intro}}}}{\dfrac{\mathsf{alt} \sim \mathsf{g}\ \mathsf{alt}}{\mathsf{alt} \sim \mathsf{g}\ \mathsf{alt}}\ \mathsf{cofix}\ \pi}\ {\sim_{\mathsf{intro}}} \tag{d}$$

We explain the proof tree in a bottom-up fashion. By the rule cofix π we first introduce the coinduction hypothesis $\pi : \mathsf{alt} \sim \mathsf{g}\ \mathsf{alt}$ into the context.[3] Next we apply the constructor $\sim_{\mathsf{intro}}$ twice. Here we have not displayed proofs d_0 and d_1 of type $\mathsf{head}\ \mathsf{alt} = \mathsf{head}\ (\mathsf{g}\ \mathsf{alt})$ and $\mathsf{head}\ (\mathsf{tail}\ \mathsf{alt}) = \mathsf{head}\ (\mathsf{tail}\ (\mathsf{g}\ \mathsf{alt}))$, respectively; both are obtained by plain equational reasoning and cause no problem. The point we want to make concerns the subproof d'' of $\mathsf{tail}^2\ \mathsf{alt} \sim \mathsf{tail}^2\ (\mathsf{g}\ \mathsf{alt})$. Left- and right-hand side can be converted by rewriting, using the hypothesis π and the assumptions (3), as follows:

$$\begin{aligned}&\mathsf{tail}^2\ \mathsf{alt} \overset{\mathsf{alt}}{\to} \mathsf{tail}^2\ (0 :: 1 :: \mathsf{alt}) \overset{\mathsf{tail}}{\to} \cdot \overset{\mathsf{tail}}{\to} \mathsf{alt} \overset{\pi}{\to} \mathsf{g}\ \mathsf{alt}\\ &\quad \overset{\mathsf{tail}}{\leftarrow} \cdot \overset{\mathsf{tail}}{\leftarrow} \mathsf{tail}^2\ (0 :: 1 :: (\mathsf{g}\ \mathsf{alt})) \overset{\mathsf{g}}{\leftarrow} \cdot \overset{\mathsf{g}}{\leftarrow} \mathsf{tail}^2\ (\mathsf{g}\ (0 :: 1 :: \mathsf{alt})) \overset{\mathsf{alt}}{\leftarrow} \mathsf{tail}^2\ (\mathsf{g}\ \mathsf{alt})\end{aligned}$$

This conversion gives rise to the proof tree d'' of the form:

$$\dfrac{e_1 : \mathsf{tail}^2\ \mathsf{alt} \sim \mathsf{alt} \qquad \dfrac{\pi : \mathsf{alt} \sim \mathsf{g}\ \mathsf{alt} \qquad e_2 : \mathsf{g}\ \mathsf{alt} \sim \mathsf{tail}^2\ (\mathsf{g}\ \mathsf{alt})}{\mathsf{alt} \sim \mathsf{tail}^2\ (\mathsf{g}\ \mathsf{alt})}\ {\sim_{\mathsf{trans}}}}{\mathsf{tail}^2\ \mathsf{alt} \sim \mathsf{tail}^2\ (\mathsf{g}\ \mathsf{alt})}\ {\sim_{\mathsf{trans}}} \tag{d''}$$

The omitted proofs e_1 and e_2 are obtained by equational reasoning without the use of the coinduction hypothesis π. But let us reconsider the proof tree d with d'' filled in by the concrete subtree above. Note that the corecursive call (or coinduction hypothesis) π is not a direct argument of the guarding constructor $\sim_{\mathsf{intro}}$, but nested within applications of $\sim_{\mathsf{trans}}$, as becomes even more apparent in the corresponding proof term[4]:

$$d = \mathsf{cofix}\ \pi\ (\sim_{\mathsf{intro}}\ d_0\ (\sim_{\mathsf{intro}}\ d_1\ (\sim_{\mathsf{trans}}\ e_1\ (\sim_{\mathsf{trans}}\ \pi\ e_2)))) \tag{4}$$

This corecursive construction is not guarded and Coq rejects it. However the term d is productive, and reduces to a constructor normal form in the limit.

Guardedness problems occur in the vast majority of cases when coinduction is combined with equational reasoning. The transformation of (productive) proof terms into guarded proof terms is the main topic of our paper. For this purpose we adopt techniques from process algebra and employ the method of 'bisimulation-up-to' [18], a generalization of Milner's bisimulation up to bisimilarity [16]. Let R be a binary relation on streams, and $\mathcal{U}$ a function from relations to relations. Then R is a bisimulation up to $\mathcal{U}$ if $\langle s, t\rangle \in R$ implies $\mathsf{head}\ s = \mathsf{head}\ t$

[3] Of course, now concluding the proof by π immediately yields a non-productive term, and is rightfully rejected by the guardedness checker.

[4] In a form analogous to stream definitions (1) and (2), the bisimilarity proof term (4) can also be written as $d = \sim_{\mathsf{intro}}\ d_0\ (\sim_{\mathsf{intro}}\ d_1\ (\sim_{\mathsf{trans}}\ e_1\ (\sim_{\mathsf{trans}}\ d\ e_2)))$.

and $\langle \mathsf{tail}\ s, \mathsf{tail}\ t\rangle \in \mathcal{U}(R)$. Under certain conditions, the fact that a relation R is a bisimulation up to $\mathcal{U}$ is sufficient to conclude that R is a subrelation of $\sim$, and so stream terms related by R are bisimilar. Typically R is included in $\mathcal{U}(R)$, and thus, in comparison with a full bisimulation, less diagrams have to be checked.

For the purpose of formalizing circular coinduction in Coq, we take $\mathcal{U}(R)$ to be the least relation including R and $\sim$, and closed under causal functions, transitivity and symmetry. A stream function F is *causal* (called 'special' in [15]) if the first n elements of the resulting stream $\mathsf{F}\ s$ only depend on the first n elements of the argument stream s. For the validity of a bisimilarity proof, it does not harm to use the coinduction hypothesis under a causal function [15].

We show soundness of the bisimulation-up-to method for this mapping $\mathcal{U}$, that is, if R is a bisimulation up to $\mathcal{U}$, then $\mathcal{U}(R)$ is a bisimulation. We also show that every circular coinduction proof can be transformed, in a structure-preserving way, to a proof that R is a bisimulation up to $\mathcal{U}$, where the relation R consists of all pairs (u, v) such that $u \sim v$ is a coinduction hypothesis in the original proof.

We thereby overcome the guardedness problem. The reason is that for proving that a relation is a bisimulation-up-to there is no need for corecursion, and hence guardedness is not an issue. The corecursive construction of bisimilarity proofs is now part of the general soundness result. In order to formalize a proof by circular coinduction in Coq, one can use our translation to obtain a proof by bisimulation-up-to $\mathcal{U}$, and then apply the soundness result to obtain a (guarded) bisimilarity proof accepted by Coq.

Related Work. Danielsson [6] works around the guardedness problem for stream definitions by defining a problem-specific language where the functions that obstruct guardedness are constructors, and defining an interpreter for the language by guarded corecursion. Recent work [14] supports compositionality in coinduction proofs by what is called 'parameterized coinduction', which allows for semantic rather than syntactic guardedness checking. In the present paper we are concerned with equational reasoning, and provide a systematic way for a Coq formalization of proofs by circular coinduction.

Overview. Sections 2 and 3 form a step-up to the main topic treated in Section 4. This order chronologically reflects how we came about to use up-to techniques. Sections 2 and 3 provide an informal discussion of how to overcome guardedness problems for several examples. In particular, in Section 2 we show how to define morphic sequences in Coq, using the idea explained above: postpone computation (viz. iterations of the morphism) in favour of guarding the corecursive call. The same idea is used in Section 3: to ensure guardedness of bisimilarity proofs, applications of transitivity are postponed by reformulating a goal $s \sim t$ into the equivalent statement $\forall s', t'.\ s' \sim s \Rightarrow t \sim t' \Rightarrow s' \sim t'$. In Section 4 we give a proof system for circular coinduction, restricted to the setting of streams over a two-element alphabet. We define our notion of bisimulation up-to $\mathcal{U}$, and discuss the soundness proof which states that if R is a bisimulation-up-to $\mathcal{U}$ then $\mathcal{U}(R)$ is a bisimulation. Finally we show that every proof by circular coinduction can be transformed to a bisimilarity proof accepted by Coq. The supporting Coq development is available as [10]. We conclude in Section 6.

2 Morphic Sequences in Coq

We show how to define morphic sequences by means of guarded corecursion. A morphic sequence (typically) is an infinite sequence obtained as the iterative fixed point of a morphism (also called a 'substitution'). Morphisms for transforming and generating infinite words provide a fundamental tool for formal languages, and have been studied extensively; we refer to [3]. We first give a standard definition of morphic sequences. Then we provide a 'direct' corecursive definition, using a productive version of the fixed point equation $h(w) = w$. Finally we show how to turn such an equation into a definition by guarded corecursion, and prove that the thus defined sequence is indeed the unique fixed point of h.

A *morphism* is a map $h : A^* \to B^*$, with A and B finite alphabets, such that $h(\varepsilon) = \varepsilon$ and $h(uv) = h(u)h(v)$ for all words $u, v \in A^*$, and can thus be defined by giving its values on the symbols of A.[5]

Let $h : A^* \to A^*$ be a morphism *prolongable* on the letter $a_0 \in A$, that is, $h(a_0) = a_0x$ for some $x \in A^*$ such that $h^i(x) \neq \varepsilon$ for all $i \geq 0$. Then we see that $h^{i+1}(a_0) = h^i(h(a_0)) = h^i(a_0x) = h^i(a_0)h^i(x)$, and hence $h^i(a_0)$ is a strict prefix of $h^{i+1}(a_0)$, for all $i \geq 0$. So then $\lim_{i\to\infty} h^i(a_0)$ exists and is infinite; this limit is denoted by $h^\omega(a_0) = \lim_{i\to\infty} h^i(a_0) = a_0\, x\, h(x) h^2(x)\, h^3(x) \cdots$ and it is readily seen to be the unique fixed point of h that starts with the letter a_0, that is, $h(h^\omega(a_0)) = h^\omega(a_0)$. A sequence $w \in A^\omega$ is *(purely) morphic* if $w = h^\omega(a_0)$ for some morphism h and starting letter a_0.

Without loss of generality [3], we may assume morphisms to be non-erasing, i.e., $h(a) \neq \varepsilon$ for all $a \in A$. Hence, we replace the condition that h be prolongable on starting letter a_0, by the simpler $h(a_0) = a_0x$ for some non-empty word x. Now we can define h as a function in $A^\omega \to A^\omega$ by guarded corecursion and pattern matching, as follows; for all $b \in A$ and $u \in A^\omega$:

$$h(b :: u) = b_0 :: b_1 :: \ldots :: b_{k-1} :: h(u) \qquad \text{where } b_0 b_1 \cdots b_{k-1} = h(b),$$

We now give a productive (yet unguarded) definition of $w = h^\omega(a)$. This method is based on the work [7,8]. Clearly, the fixed point equation $w = h(w)$, where we now view w as a recursion variable, is not productive (and, typically, also does not have a unique solution for w). By using the knowledge that the first letter of w is a_0, we can turn it into

$$w = a_0 :: w' \qquad\qquad w' = \mathit{tail}(h(w)) \tag{5}$$

In order to see that this specification is productive indeed, we plug in the information that $h(a_0) = a_0x$ where, say, $x = a_1a_2 \cdots a_k$ with $k \geq 1$. We do so by replacing w by $a_0 :: w'$, in the right-hand side of the equation for w' and rewriting h and *tail*; we then obtain

$$w = a_0 :: w' \qquad\qquad w' = a_1 :: a_2 :: \ldots :: a_k :: h(w') \tag{6}$$

This is clearly a productive equation, because h 'consumes' one stream element at most, and, being non-erasing, produces one stream element at least.

[5] Juxtaposition of words denotes concatenation.

However, as will be clear by now, the corecursive equation for w' is not guarded, because the recursive call is nested within h.

In order to solve this problem, we generalize the construction by adding an argument on which h is applied, and from which we can always extract the next element to produce. The construction is reminiscent of Emile Post's tag systems [17]. Let the type of our morphism h be $A^+ \to A^+$, mapping nonempty words to nonempty words. Then we can define the morphic stream $w = h^\omega(a_0)$ by

$$w = a_0 :: tag_h(x) \qquad\qquad tag_h(ay) = a :: tag_h(yh(a)) \qquad (7)$$

where we recall that the morphism h, the starting letter a_0 and the non-empty word x are such that $h(a_0) = a_0x$. We note that the function $tag_h : A^+ \to A^\omega$ is defined by guarded corecursion, and hence (7) is accepted by Coq.

In this general set-up we prove (in Coq, see [10]) that the stream w defined by (7) is indeed the fixed point of h (and so satisfies also the equations (5) and (6)). That w is the unique fixed point (starting with a_0) follows from the fact that h is non-erasing and prolonging on a_0.

Example 1. The Fibonacci word $0100101001001\cdots$ [3] is generated by iterating the morphism f defined by $f(0) = 01$ and $f(1) = 0$, on the starting letter 0. In Coq we define nonempty lists inductively, we use $[a]$ for the singleton list, and overload the symbol :: to also denote the constructor for nonempty lists. The Fibonacci word fib is thus defined by

$$\begin{array}{lll}
\mathsf{fib} = 0 :: \mathsf{tag_f}\ [1] & \mathsf{tag_f}\ [a] = a :: \mathsf{tag_f}\ (\mathsf{f}\ a) & \mathsf{tag_f}\ (a :: u) = a :: \mathsf{tag_f}\ (u +\!\!+ \mathsf{f}\ a) \\
\mathsf{f}\ [0] = 0 :: [1] & \mathsf{f}\ (0 :: u) = 0 :: 1 :: \mathsf{f}\ u & [a] +\!\!+ v = a :: v \\
\mathsf{f}\ [1] = [0] & \mathsf{f}\ (1 :: u) = 0 :: \mathsf{f}\ u & (a :: u) +\!\!+ v = x :: (u +\!\!+ v)
\end{array}$$

3 Coinduction Loading

The idea of guarding the corecursive call by hiding the computation in an extra argument, as outlined in the previous section, turns out to be useful in the setting of bisimilarity proofs as well. We present the method of *coinduction loading*, which in some cases suffices to turn a productive bisimilarity proof into a guarded one. Here we want to avoid that transitivity of $\sim$ is applied to the coinduction hypothesis (= corecursive call). The idea is to reformulate a goal $s \sim t$ into the equivalent statement

$$\forall s', t'.\ s' \sim s \Rightarrow t \sim t' \Rightarrow s' \sim t'\,.$$

This enables us to move applications of transitivity to the argument of the corecursive call, as illustrated by the following example. Suppose we are given the following definitions (=) and assumption ($\sim$):

$$\begin{array}{ll}
\mathsf{dup}\ (x :: s) = x :: x :: \mathsf{dup}\ s & \mathsf{exp}\ (x :: s) \sim x :: \mathsf{dup}\ (\mathsf{exp}\ s) \\
\mathsf{odd}\ (x :: y :: s) = y :: \mathsf{odd}\ s & \mathsf{log}\ (x :: s) = x :: \mathsf{log}\ (\mathsf{odd}\ s)
\end{array}$$

The behavior of these functions is illustrated by applying them to the stream $\mathsf{nats} = 0 :: 1 :: 2 :: \ldots$:

$$\mathsf{dup}\ \mathsf{nats} = 0 :: 0 :: 1 :: 1 :: 2 :: 2 :: \ldots \qquad \mathsf{exp}\ \mathsf{nats} = 0 :: 1 :: 1 :: 2 :: 2 :: 2 :: 2 :: \ldots$$
$$\mathsf{odd}\ \mathsf{nats} = 1 :: 3 :: 5 :: 7 :: 9 :: \ldots \qquad \mathsf{log}\ \mathsf{nats} = 0 :: 2 :: 6 :: 14 :: 30 :: 62 :: \ldots$$

The assumption for exp can, in Coq, not be taken as a definition, because the corecursive call is nested within dup, and so is not guarded, although certainly productive. On the other hand, the corecursive equation for log is perfectly guarded. The function log has a logarithmically increasing production function [7], i.e., n elements in leads to $\lfloor \log_2(n+1) \rfloor$ elements out. Definitions are always preferable to assumptions since Coq's native evaluation does not harm guardedness of proofs, whereas rewriting terms by using bisimilarity proofs does. Consider the following corecursive proof term[6], which witnesses that odd ∘ dup is the identity function:

$$\mathsf{cofix}\ \pi\ (\lambda(x :: s').\ \sim_{\mathsf{intro}}\ (=_{\mathsf{refl}}\ x)\ (\pi\ s'))\ :\ \forall s.\ \mathsf{odd}\ (\mathsf{dup}\ s) \sim s \tag{8}$$

Note that this proof term is defined by guarded corecursion. After we have destructed s into $x :: s'$, the terms tail (odd (dup $(x :: s')$)) and odd (dup s') are convertible by Coq's native evaluation; similarly the subgoal head (odd (dup $(x :: s')$)) $= x$ is proved by reflexivity. Finally, the application of the coinduction hypothesis $\pi : \forall s.\ \mathsf{odd}\ (\mathsf{dup}\ s) \sim s$ to s' proves that odd (dup s') is bisimilar to s'.

Now we want to prove that also the composition log ∘ exp is the identity:

$$\forall s.\ \mathsf{log}\ (\mathsf{exp}\ s) \sim s \tag{9}$$

The proof that we want to construct (but which is not accepted by Coq) looks as follows; here we have omitted the subterms d of type head (log (exp $(x :: s')$)) $= x$, and e whose type is indicated in the tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{e : \mathsf{tail}\ (\mathsf{log}\ (\mathsf{exp}\ (x :: s'))) \sim \mathsf{log}\ (\mathsf{exp}\ s') \qquad \pi\ s' : \mathsf{log}\ (\mathsf{exp}\ s') \sim s'}{\mathsf{tail}\ (\mathsf{log}\ (\mathsf{exp}\ (x :: s'))) \sim s'}\ \sim_{\mathsf{trans}}}{\mathsf{log}\ (\mathsf{exp}\ (x :: s')) \sim x :: s'}\ \sim_{\mathsf{intro}}}{\forall s.\ \mathsf{log}\ (\mathsf{exp}\ s) \sim s}\ \lambda(x :: s')}{\forall s.\ \mathsf{log}\ (\mathsf{exp}\ s) \sim s}\ \mathsf{cofix}\ \pi}{d}$$

Guardedness of the corecursive call $\pi\ s'$ is here obstructed by the application of $\sim_{\mathsf{trans}}$, transitivity of $\sim$. We explain why we cannot do without $\sim_{\mathsf{trans}}$ in this case. In order to prove that tail (log (exp $(x :: s')$)) is bisimilar to s' we cannot use Coq's native evaluation, for example because exp is not a defined function. Instead we proceed by rewriting using the coinduction hypothesis $\pi\ s'$, the assumption for exp and the lemma (8), as follows:

$$\mathsf{tail}\ (\mathsf{log}\ (\mathsf{exp}\ (x :: s'))) \overset{\mathsf{exp}}{\to} \mathsf{tail}\ (\mathsf{log}\ (x :: \mathsf{dup}\ (\mathsf{exp}\ s'))) \overset{\mathsf{log}}{\to}$$
$$\mathsf{tail}\ (x :: \mathsf{log}\ (\mathsf{odd}\ (\mathsf{dup}\ (\mathsf{exp}\ s')))) \overset{\mathsf{tail}}{\to} \mathsf{log}\ (\mathsf{odd}\ (\mathsf{dup}\ (\mathsf{exp}\ s'))) \overset{(8)}{\to} \mathsf{log}\ (\mathsf{exp}\ s') \overset{\pi}{\to} s'$$

All these rewrite steps are connected by applications of $\sim_{\mathsf{trans}}$. The proof tree above arises from splitting this rewrite sequence at the term log (exp s') (the middle term in the displayed application of $\sim_{\mathsf{trans}}$). In whatever way we split the sequence, the coinduction hypothesis is argument of at least one application of $\sim_{\mathsf{trans}}$, resulting in an unguarded proof term.

[6] We use the syntax $\lambda(x :: s').\ t$ to denote abstraction and pattern matching at once.

This can be fixed with coinduction loading, thereby turning the above proof of (9) into a guarded proof of the equivalent statement $\forall s, t_1, t_2.(t_1 \sim \mathsf{log}\ (\mathsf{exp}\ s)\ \Rightarrow\ s \sim t_2\ \Rightarrow\ t_1 \sim t_2)$, as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{d' \qquad (\pi\ s'\ (\mathsf{tail}\ t_1)\ (\mathsf{tail}\ t_2)\ d_1\ d_2) : \mathsf{tail}\ t_1 \sim \mathsf{tail}\ t_2}{t_1 \sim t_2}\ \sim_{\mathsf{intro}}}{t_1 \sim \mathsf{log}\ (\mathsf{exp}\ (x :: s')) \ \Rightarrow\ x :: s' \sim t_2 \ \Rightarrow\ t_1 \sim t_2}\ \lambda\gamma_1, \gamma_2}{\forall s, t_1, t_2.\ (t_1 \sim \mathsf{log}\ (\mathsf{exp}\ s) \ \Rightarrow\ s \sim t_2 \ \Rightarrow\ t_1 \sim t_2)}\ \lambda(x :: s'), t_1, t_2}{\forall s, t_1, t_2.\ (t_1 \sim \mathsf{log}\ (\mathsf{exp}\ s) \ \Rightarrow\ s \sim t_2 \ \Rightarrow\ t_1 \sim t_2)}\ \mathsf{cofix}\ \pi$$

We note that λ-abstractions (introductions of $\forall$ and $\Rightarrow$) do not obstruct guardedness. We ignore the term $d' : \mathsf{head}\ t_1 = \mathsf{head}\ t_2$ which is a modification of $d : \mathsf{head}\ (\mathsf{log}\ (\mathsf{exp}\ (x :: s'))) = x$ using the hypotheses $\gamma_1 : t_1 \sim \mathsf{log}\ (\mathsf{exp}\ (x :: s'))$ and $\gamma_2 : s \sim t_2$. The other subtrees, d_1 and d_2, are given by the trees below. Here $\mathsf{comp}_{\mathsf{tail}}$ is an instance of $\mathsf{comp}_{\mathsf{f}}$ referring to the compatibility of a unary stream function f, i.e., $\mathsf{f}\ s \sim \mathsf{f}\ t$ whenever $s \sim t$. In Coq, compatibility cannot be proved for arbitrary f, but for concrete instances this forms no problem.

$$\dfrac{\dfrac{\gamma_1 : t_1 \sim \mathsf{log}\ (\mathsf{exp}\ (x :: s'))}{\mathsf{tail}\ t_1 \sim \mathsf{tail}\ (\mathsf{log}\ (\mathsf{exp}\ (x :: s')))}\ \mathsf{comp}_{\mathsf{tail}} \qquad e}{d_1 : \mathsf{tail}\ t_1 \sim \mathsf{log}\ (\mathsf{exp}\ s')}\ \sim_{\mathsf{trans}} \qquad \dfrac{\gamma_2 : x :: s' \sim t_2}{d_2 : s' \sim \mathsf{tail}\ t_2}\ \mathsf{comp}_{\mathsf{tail}}$$

Instead of giving a more formal definition of the transformation suggested by the above example, we continue our exposition with incorporating bisimulation-up-to techniques. The transformation described above will turn out to be an instance of the theory of bisimulation-up-to, namely as bisimulations up to transitivity and bisimilarity.

4 Circular Coinduction

We introduce a proof system for circular coinduction [12,15,19,21]. For the sake of presentation, we focus on streams over $\{0, 1\}$, but it is straightforward to generalize the method to infinite terms (ranked trees).

We assume that the signature is declared in Coq. We emphasize that the terms introduced below are just a notation for Coq terms. We have sorts $\mathfrak{B}$ and $\mathfrak{S}$ for $\{0, 1\}$ and streams over $\{0, 1\}$, respectively. A *signature* Σ is a set of symbols each having a fixed *type* in $\{\mathfrak{B}, \mathfrak{S}\}^* \times \{\mathfrak{B}, \mathfrak{S}\}$. We write $f : t_1 \times \ldots \times t_n \to s$ whenever $f \in \Sigma$ has type $\langle\langle t_1, \ldots, t_n\rangle, s\rangle$. Let Σ be a signature, and $\mathcal{X} = \mathcal{X}_{\mathfrak{B}} \cup \mathcal{X}_{\mathfrak{S}}$ be a set of variables such that $\mathcal{X} \cap \Sigma = \varnothing$. The set of *data terms* $T_{\mathfrak{B}}$ and *stream terms* $T_{\mathfrak{S}}$ over Σ and $\mathcal{X}$ are inductively defined by the grammar:

$$T_s ::= x \mid f(T_{s_1}, \ldots, T_{s_n}) \qquad (x \in \mathcal{X}_s,\ f \in \Sigma,\ f : s_1 \times \ldots \times s_n \to s)$$

for $s \in \{\mathfrak{B}, \mathfrak{S}\}$. We write T for $T_{\mathfrak{B}} \cup T_{\mathfrak{S}}$. A *substitution* is a mapping $\sigma : \mathcal{X} \to T$ that respects the sorts, that is, $\sigma(x) \in T_{\mathfrak{B}}$ for every $x \in \mathcal{X}_{\mathfrak{B}}$, and $\sigma(x) \in T_{\mathfrak{S}}$ for every $x \in \mathcal{X}_{\mathfrak{S}}$. We write $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ to abbreviate the substitution

defined by $\sigma(x_1) = t_1, \ldots, \sigma(x_n) = t_n$ and $\sigma(x) = x$ for every $x \notin \{x_1, \ldots, x_n\}$. For terms $s \in T$ and substitutions σ, we define s^σ inductively by $f(t_1, \ldots, t_n)^\sigma = f(t_1^\sigma, \ldots, t_n^\sigma)$ and $x^\sigma = \sigma(x)$. A *stream context* C is a term of sort $\mathfrak{S}$ over Σ and $\mathcal{X} \cup \{\Box\}$ where $\Box$ is a fresh variable of sort $\mathfrak{S}$. For terms $s \in T_{\mathfrak{S}}$ and contexts C, we write $C[s]$ for the term $C^{\Box \mapsto s}$.

We define *bisimilarity up to depth n* on stream terms inductively as follows:

$$\frac{}{s \sim_0 t} \qquad \frac{\mathsf{head}\ s = \mathsf{head}\ t \qquad \mathsf{tail}\ s \sim_n \mathsf{tail}\ t}{s \sim_{n+1} t}$$

A *causal context* is a context C such that for all stream terms s, t we have:

$$s \sim_n t \ \Rightarrow\ C[s] \sim_n C[t]\,, \quad \text{for all } n \in \mathbb{N}.$$

Examples of causal contexts are $\mathsf{dup}\ \Box$, $\mathsf{exp}\ \Box$, and $\mathsf{log}\ (\mathsf{exp}\ \Box)$; examples of non-causal contexts are $\mathsf{tail}\ \Box$, $\mathsf{odd}\ \Box$, and $\mathsf{log}\ \Box$, (see previous section). Note that for every causal context C we have that $\mathsf{tail}\ C[a :: \Box]$ is again causal.

Definition 2. The set Π of *(circular coinduction) proof terms* is inductively defined as follows (the superscript 'cc' stands for circular coinduction):

$$\begin{aligned}\Pi \ ::= \sim^{\mathsf{cc}}_{\mathsf{hyp}}\ \gamma\ C\ \sigma \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{cut}}\ \gamma\ \Pi\ \Pi \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{cohyp}}\ \delta\ D\ \sigma \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta\ E\ \Pi \ \mid\\ \sim^{\mathsf{cc}}_{\mathsf{refl}}\ s \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{sym}}\ \Pi \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{trans}}\ \Pi\ \Pi \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{B}}}\ x\ \Pi\ \Pi \ \mid\ \sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{S}}}\ y\ \Pi\end{aligned}$$

where γ, δ are names for hypotheses, $s \in T_{\mathfrak{S}}$ is a stream term, $x \in \mathcal{X}_{\mathfrak{B}}$ and $y \in \mathcal{X}_{\mathfrak{S}}$ are variables of type $\mathfrak{B}$ and $\mathfrak{S}$, respectively, $\sigma : \mathcal{X} \to T$ is a substitution, and $C, D \in T$ are stream contexts, with D causal. The class E of equational proofs on data terms (equality of the heads) is left implicit.

The constructor $\sim^{\mathsf{cc}}_{\mathsf{coin}}$ represents the combination of the Coq constructs cofix and $\sim_{\mathsf{intro}}$ introduced earlier. We leave universal quantification implicit. That is, for stream terms s, t that contain variables $x_1, \ldots, x_n$, and for a relation R on stream terms, we write $s\ R\ t$ to denote $\forall x_1, \ldots, x_n.\ s\ R\ t$.

We now define 'typing judgments' for proof terms depending on two proof contexts Γ and Δ, consisting of triples written as $\gamma : s \sim^{\mathsf{cc}} t$. Here Γ contains assumptions, and Δ contains the coinduction hypotheses that are introduced in the construction of the proof. The intuition is that $\Gamma, \Delta \vdash d : s \sim^{\mathsf{cc}} t$ means that for all $n \in \mathbb{N}$, if all pairs in Γ are (fully) bisimilar, and all pairs in Δ are bisimilar up to depth n, then s is bisimilar to t up to depth $n+1$. The semantics for judgments, will be given in Section 5 where we translate them into Coq bisimilarity.

Definition 3. Let Γ, Δ be sets of triples $\gamma : u \sim^{\mathsf{cc}} v$, where every name γ appears at most once in $\Gamma \cup \Delta$. Let $d \in \Pi$ be a proof term, and s, t stream terms. We define the *judgment* $\Gamma, \Delta \vdash d : s \sim^{\mathsf{cc}} t$ inductively by the rules in Figure 1.

Next we introduce rewriting as a syntax for constructing proof terms.

Definition 4. Let $\gamma : u \sim^{\mathsf{cc}} v \in \Gamma$ and $\delta : u \sim^{\mathsf{cc}} v \in \Delta$. Let C be a context, σ a substitution, and abbreviate $s = C[u^\sigma]$ and $t = C[v^\sigma]$. Then we define

$$s \xrightarrow{\gamma} t = (\sim^{\mathsf{cc}}_{\mathsf{hyp}}\ \gamma\ C\ \sigma) \qquad s \xleftarrow{\gamma} t = (\sim^{\mathsf{cc}}_{\mathsf{sym}}\ (\sim^{\mathsf{cc}}_{\mathsf{hyp}}\ \gamma\ C\ \sigma))$$

$$\frac{}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{hyp}}\ \gamma\ C\ \sigma) : C[s^\sigma] \sim^{\mathsf{cc}} C[t^\sigma]}\ (\gamma : s \sim^{\mathsf{cc}} t) \in \Gamma$$

$$\frac{}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{cohyp}}\ \delta\ D\ \sigma) : D[s^\sigma] \sim^{\mathsf{cc}} D[t^\sigma]}\ (\delta : s \sim^{\mathsf{cc}} t) \in \Delta,\ D \text{ is causal}$$

$$\frac{}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{refl}}\ s) : s \sim^{\mathsf{cc}} s} \qquad \frac{\Gamma, \Delta \vdash d : t \sim^{\mathsf{cc}} s}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{sym}}\ d) : s \sim^{\mathsf{cc}} t}$$

$$\frac{\Gamma, \Delta \vdash d_1 : s \sim^{\mathsf{cc}} u \quad \Gamma, \Delta \vdash d_2 : u \sim^{\mathsf{cc}} t}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{trans}}\ d_1\ d_2\ ) : s \sim^{\mathsf{cc}} t}$$

$$\frac{\Gamma \vdash d_0 : (\mathsf{head}\ s = \mathsf{head}\ t) \quad \Gamma, \Delta \cup \{\delta : s \sim^{\mathsf{cc}} t\} \vdash d' : (\mathsf{tail}\ s \sim^{\mathsf{cc}} \mathsf{tail}\ t)}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta\ d_0\ d') : s \sim^{\mathsf{cc}} t}\ \delta \notin \Delta$$

$$\frac{\Gamma, \Delta \vdash d_0 : (s \sim^{\mathsf{cc}} t)^{x \mapsto 0} \quad \Gamma, \Delta \vdash d_1 : (s \sim^{\mathsf{cc}} t)^{x \mapsto 1}}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{B}}}\ x\ d_0\ d_1) : s \sim^{\mathsf{cc}} t}\ x \in \mathcal{X}_{\mathfrak{B}}$$

$$\frac{\Gamma, \Delta \vdash d : (s \sim^{\mathsf{cc}} t)^{x \mapsto y::z}}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{S}}}\ x\ d) : s \sim^{\mathsf{cc}} t}\ y \in \mathcal{X}_{\mathfrak{B}},\ z \in \mathcal{X}_{\mathfrak{S}} \text{ fresh for } s, t; \text{ and } x \in \mathcal{X}_{\mathfrak{S}}$$

$$\frac{\Gamma \cup \{\gamma : u \sim^{\mathsf{cc}} v\}, \Delta \vdash d_1 : s \sim^{\mathsf{cc}} t \quad \Gamma, \varnothing \vdash d_2 : u \sim^{\mathsf{cc}} v}{\Gamma, \Delta \vdash (\sim^{\mathsf{cc}}_{\mathsf{cut}}\ \gamma\ d_1\ d_2) : s \sim^{\mathsf{cc}} t}\ \gamma \notin \Gamma$$

Fig. 1. Proof rules for circular coinduction for stream terms

$$s \xrightarrow{\delta} t = (\sim^{\mathsf{cc}}_{\mathsf{cohyp}}\ \delta\ C\ \sigma) \qquad\qquad s \xleftarrow{\delta} t = (\sim^{\mathsf{cc}}_{\mathsf{sym}}\ (\sim^{\mathsf{cc}}_{\mathsf{cohyp}}\ \delta\ C\ \sigma))$$

where in the case of $\xrightarrow{\delta}$ and $\xleftarrow{\delta}$, C is additionally required to be a causal context. Furthermore, let $\Xi = \{\xrightarrow{\gamma}, \xleftarrow{\gamma} \mid \gamma \in \Gamma\} \cup \{\xrightarrow{\delta}, \xleftarrow{\delta} \mid \delta \in \Delta\}$. Then for every $\leftrightarrow \in \Xi$ the judgment $\Gamma, \Delta \vdash s \leftrightarrow t : s \sim^{\mathsf{cc}} t$ holds. We define $s_0 \leftrightarrow_1 s_1 \leftrightarrow_2 \ldots \leftrightarrow_n s_n$ inductively by $(\sim^{\mathsf{cc}}_{\mathsf{trans}}\ (s_0 \leftrightarrow_1 s_1)\ (s_1 \leftrightarrow_2 \ldots \leftrightarrow_n s_n)\ )$ where $\leftrightarrow_i \in \Xi$ for $1 \le i \le n$.

For uniformity we work with assumptions only, but functions specified by guarded corecursion (like log in Section 3) can of course be taken as definitions in Coq, and then rewriting comes 'for free', i.e., are not reflected in the proof tree.

The following example illustrates the use of the syntax of Definition 4.

Example 5. Given the assumptions[7] $\Gamma = \{\gamma_1 : z_1 \sim^{\mathsf{cc}} (0 :: z_2), \gamma_2 : z_2 \sim^{\mathsf{cc}} (0 :: z_1)\}$, our goal is to construct a proof term witnessing $z_1 \sim^{\mathsf{cc}} z_2$. As usual we first apply the $\sim^{\mathsf{cc}}_{\mathsf{coin}}$-rule, i.e., we assume what we have to prove as a coinduction hypothesis $\delta : z_1 \sim^{\mathsf{cc}} z_2$, and then construct terms d_0 and d' so that

$$\frac{\Gamma \vdash d_0 : \mathsf{head}\ z_1 = \mathsf{head}\ z_2 \qquad \Gamma, \{\delta : z_1 \sim^{\mathsf{cc}} z_2\} \vdash d' : \mathsf{tail}\ z_1 \sim^{\mathsf{cc}} \mathsf{tail}\ z_2}{\Gamma, \varnothing \vdash (\sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta\ d_0\ d') : z_1 \sim^{\mathsf{cc}} z_2}\ \sim^{\mathsf{cc}}_{\mathsf{coin}}$$

[7] These mutual corecursive equations can actually be taken as definitions in Coq.

By rewriting we obtain

$$d_0 \;=\; \mathsf{head}\ z_1 \overset{\gamma_1}{\to} \mathsf{head}\ (0 :: z_2) \overset{\gamma_{\mathsf{head}}}{\to} 0 \overset{\gamma_{\mathsf{head}}}{\leftarrow} \mathsf{head}\ (0 :: z_1) \overset{\gamma_2}{\leftarrow} \mathsf{head}\ z_2$$

$$d' \;=\; \mathsf{tail}\ z_1 \overset{\gamma_1}{\to} \mathsf{tail}\ (0 :: z_2) \overset{\gamma_{\mathsf{tail}}}{\to} z_2 \overset{\delta}{\leftarrow} z_1 \overset{\gamma_{\mathsf{tail}}}{\leftarrow} \mathsf{tail}\ (0 :: z_1) \overset{\gamma_2}{\leftarrow} \mathsf{tail}\ z_2$$

So the proof tree corresponding to d' is as follows, where we write s' to denote $\mathsf{tail}\ s$, and where we omit contexts and proof terms:

$$\dfrac{\dfrac{\gamma_1 : z_1 \sim^{\mathsf{cc}} (0 :: z_2)}{z_1' \sim^{\mathsf{cc}} (0 :: z_2)'}\sim^{\mathsf{cc}}_{\mathsf{hyp}} \quad \dfrac{\gamma_{\mathsf{tail}} : (x :: s)' \sim^{\mathsf{cc}} s}{(0 :: z_2)' \sim^{\mathsf{cc}} z_2}\sim^{\mathsf{cc}}_{\mathsf{hyp}} \quad \dfrac{\dfrac{\delta : z_1 \sim^{\mathsf{cc}} z_2}{z_1 \sim^{\mathsf{cc}} z_2}\sim^{\mathsf{cc}}_{\mathsf{cohyp}}}{z_2 \sim^{\mathsf{cc}} z_1}\sim^{\mathsf{cc}}_{\mathsf{sym}} \quad \dfrac{\dfrac{\gamma_{\mathsf{tail}} : (x :: s)' \sim^{\mathsf{cc}} s}{(0 :: z_1)' \sim^{\mathsf{cc}} z_1}\sim^{\mathsf{cc}}_{\mathsf{hyp}}}{z_1 \sim^{\mathsf{cc}} (0 :: z_1)'}\sim^{\mathsf{cc}}_{\mathsf{sym}} \quad \dfrac{\gamma_2 : z_2 \sim^{\mathsf{cc}} 0 :: z_1}{z_2' \sim^{\mathsf{cc}} (0 :: z_1)'}\sim^{\mathsf{cc}}_{\mathsf{hyp}}}{(0 :: z_1)' \sim^{\mathsf{cc}} z_2'}\sim^{\mathsf{cc}}_{\mathsf{sym}}}{z_1 \sim^{\mathsf{cc}} z_2'}\sim^{\mathsf{cc}}_{\mathsf{trans}}}{z_2 \sim^{\mathsf{cc}} z_2'}\sim^{\mathsf{cc}}_{\mathsf{trans}}}{(0 :: z_2)' \sim^{\mathsf{cc}} z_2'}\sim^{\mathsf{cc}}_{\mathsf{trans}}}{z_1' \sim^{\mathsf{cc}} z_2'}\sim^{\mathsf{cc}}_{\mathsf{trans}}$$

Next, we give a detailed example of a proof by circular coinduction, i.e., using the system introduced in Definition 3. In Section 5 we discuss how this proof is translated into a proof that Coq accepts.

Consider the following guarded equations (here taken as assumptions) of functions $\mathsf{D}, \mathsf{T}, +$

$$\gamma_{\mathsf{D}} \;:\; \mathsf{D}\ s \sim^{\mathsf{cc}} s + \mathsf{tail}\ s \qquad \gamma_{\mathsf{T}} \;:\; \mathsf{T}\ s \sim^{\mathsf{cc}} \mathsf{head}\ s :: \mathsf{T}\ (\mathsf{D}\ s)$$

$$\gamma_{+} \;:\; (x :: s) + (y :: t) \sim^{\mathsf{cc}} (x + y) :: (s + t)$$

where $0 + 0 = 1 + 1 = 0$ and $0 + 1 = 1 + 0 = 1$.

Our goal is to prove by circular coinduction that T is an involution, that is, $\mathsf{T}\ (\mathsf{T}\ s) \sim s$, for all stream terms s. For this we use some easily proven facts about addition, and distribution of D over $+$:

$$\gamma_{+\mathsf{ass}} \;:\; x + (y + z) \sim^{\mathsf{cc}} (x + y) + z \qquad \gamma_{+\mathsf{com}} \;:\; x + y \sim^{\mathsf{cc}} y + x$$

$$\gamma_{+\mathsf{id}} \;:\; x + \mathsf{zeros} \sim^{\mathsf{cc}} x \qquad \gamma_{+\mathsf{ann}} \;:\; x + x \sim^{\mathsf{cc}} \mathsf{zeros}$$

$$\gamma_{\mathsf{Ddistr}} \;:\; \mathsf{D}\ (x + y) \sim^{\mathsf{cc}} \mathsf{D}\ x + \mathsf{D}\ y$$

and we let $\Gamma = \{\, \gamma_{\mathsf{head}}, \gamma_{\mathsf{tail}}, \gamma_{+}, \gamma_{\mathsf{D}}, \gamma_{\mathsf{T}}, \gamma_{+\mathsf{ass}}, \gamma_{+\mathsf{com}}, \gamma_{+\mathsf{id}}, \gamma_{+\mathsf{ann}}, \gamma_{\mathsf{Ddistr}} \,\}$.

Another lemma that we need, is distributivity of T over $+$. This in turn uses distributivity of D over $+$, which in Coq would destroy guardedness by invoking transitivity. The proof of $\mathsf{T}\ (x + y) \sim^{\mathsf{cc}} \mathsf{T}\ x + \mathsf{T}\ y$ has the following shape:

$$\dfrac{d_0 \qquad \dfrac{\dfrac{d' : \mathsf{tail}\ (\mathsf{T}\ ((a :: x') + (b :: y'))) \sim^{\mathsf{cc}} \mathsf{tail}\ (\mathsf{T}\ (a :: x') + \mathsf{T}\ (b :: y'))}{\mathsf{tail}\ (\mathsf{T}\ ((a :: x') + (b :: y'))) \sim^{\mathsf{cc}} \mathsf{tail}\ (\mathsf{T}\ (a :: x') + \mathsf{T}\ (b :: y'))}\sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{S}}}}{\mathsf{tail}\ (\mathsf{T}\ (x + y)) \sim^{\mathsf{cc}} \mathsf{tail}\ (\mathsf{T}\ x + \mathsf{T}\ y)}\sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{S}}}}{\mathsf{T}\ (x + y) \sim^{\mathsf{cc}} \mathsf{T}\ x + \mathsf{T}\ y}\sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta$$

We ignore the proof d_0 of $\mathsf{head}\ (\mathsf{T}\ (x + y)) \sim^{\mathsf{cc}} \mathsf{head}\ (\mathsf{T}\ x + \mathsf{T}\ y)$, but define d' by rewriting, using the coinduction hypothesis $\delta : \mathsf{T}\ (x + y) \sim^{\mathsf{cc}} \mathsf{T}\ x + \mathsf{T}\ y$:

$$\mathsf{tail}\ (\mathsf{T}\ ((a :: x') + (b :: y'))) \overset{\gamma_{+}}{\to}\overset{\gamma_{\mathsf{T}}}{\to}\overset{\gamma_{\mathsf{tail}}}{\to} \mathsf{T}\ (\mathsf{D}\ (x' + y')) \overset{\gamma_{\mathsf{Ddistr}}}{\to} \mathsf{T}\ (\mathsf{D}\ x' + \mathsf{D}\ y')$$

$$\xrightarrow{\delta} \mathsf{T}\ (\mathsf{D}\ x') + \mathsf{T}\ (\mathsf{D}\ y') \xleftarrow{\gamma_{\mathsf{tail}}}\xleftarrow{\gamma_{+}}\xleftarrow{\gamma_{\mathsf{T}}}\xleftarrow{\gamma_{\mathsf{T}}} \mathsf{tail}\ (\mathsf{T}\ (a :: x') + \mathsf{T}\ (b :: y'))$$

Thus we have shown that the following judgment holds using the proof system for circular coinduction given in Figure 1:

$$\Gamma, \varnothing \vdash (\sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta\ (\sim^{\mathsf{cc}}_{\mathsf{case}_{\mathbb{S}}}\ x\ (\sim^{\mathsf{cc}}_{\mathsf{case}_{\mathbb{S}}}\ y\ d'))) : \mathsf{T}\ (x + y) \sim^{\mathsf{cc}} \mathsf{T}\ x + \mathsf{T}\ y$$

We now continue with showing that T is an involution. Let $\Gamma' = \Gamma \cup \{\gamma_{\mathsf{Tdistr}} : \mathsf{T}\ (x + y) \sim^{\mathsf{cc}} \mathsf{T}\ x + \mathsf{T}\ y\}$, i.e., we take up the lemma we have just proved as an assumption. The proof again starts by:

$$\frac{e_0 : \mathsf{head}\ (\mathsf{T}\ (\mathsf{T}\ s)) \sim^{\mathsf{cc}} \mathsf{head}\ s \qquad e' : \mathsf{tail}\ (\mathsf{T}\ (\mathsf{T}\ s)) \sim^{\mathsf{cc}} \mathsf{tail}\ s}{\mathsf{T}\ (\mathsf{T}\ s) \sim^{\mathsf{cc}} s}\ \sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta$$

and the proof term e' witnessing bisimilarity of $\mathsf{tail}\ (\mathsf{T}\ (\mathsf{T}\ s))$ and $\mathsf{tail}\ s$ is constructed by the following rewrite sequence:

$$\mathsf{tail}\ (\mathsf{T}\ (\mathsf{T}\ s)) \xrightarrow{\gamma_{\mathsf{T}}} \cdot \xrightarrow{\gamma_{\mathsf{tail}}} \mathsf{T}\ (\mathsf{D}\ (\mathsf{T}\ s)) \xrightarrow{\gamma_{\mathsf{D}}} \mathsf{T}\ (\mathsf{T}\ s + \mathsf{tail}\ (\mathsf{T}\ s)) \xrightarrow{\gamma_{\mathsf{T}}} \cdot \xrightarrow{\gamma_{\mathsf{tail}}} \mathsf{T}\ (\mathsf{T}\ s + \mathsf{T}\ (\mathsf{D}\ s))$$
$$\xrightarrow{\gamma_{\mathsf{Tdistr}}} \mathsf{T}\ (\mathsf{T}\ s) + \mathsf{T}\ (\mathsf{T}\ (\mathsf{D}\ s)) \xrightarrow{\delta} s + \mathsf{T}\ (\mathsf{T}\ (\mathsf{D}\ s)) \xrightarrow{\delta} s + \mathsf{D}\ s \xrightarrow{\gamma_{\mathsf{D}}} s + (s + \mathsf{tail}\ s)$$
$$\xrightarrow{\gamma_{+\mathsf{ass}}} (s + s) + \mathsf{tail}\ s \xrightarrow{\gamma_{+\mathsf{ann}}} \mathsf{zeros} + \mathsf{tail}\ s \xrightarrow{\gamma_{+\mathsf{id}}} \mathsf{tail}\ s$$

Here we have used the coinduction hypothesis δ twice. The first application (from left to right) under the causal context $\square + \mathsf{T}\ (\mathsf{T}\ (\mathsf{D}\ s))$, and the second under the causal context $s + \square$.

The above example illustrates several features of circular coinduction that cannot be captured by the method of coinduction loading introduced in the previous section (and certainly not by guarded corecursion). Without further generalizing, the method of coinduction loading cannot deal with more than one application of the coinduction hypothesis, and also does not allow for the use of the coinduction hypotheses under causal contexts.

The next section shows how to translate circular coinduction into Coq proofs.

5 Bisimulation-Up-To

To avoid the problems with guardedness in constructing a corecursive proof term for proving $s \sim t$, the user can instead define a relation R on stream terms with $\langle s, t\rangle \in R$, and then show that R is a bisimulation. This suffices to obtain a proof of $s \sim t$ in Coq, as follows: let $h : \forall s, t : A^\omega.\ s\ R\ t \Rightarrow \mathsf{head}\ s = \mathsf{head}\ t \wedge (\mathsf{tail}\ s)\ R\ (\mathsf{tail}\ t)$, witnessing that R is a bisimulation. A (Coq) proof term of type $\forall s, t.\ s\ R\ t \Rightarrow s \sim t$ is

$$\mathsf{cofix}\ \delta\ (\lambda s, t : A^\omega.\ \lambda\gamma : s\ R\ t.\ (\sim_{\mathsf{intro}}\ d_0\ (\delta\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)\ d')))$$

where $d_0 = \mathsf{pj}_1\ (h\ s\ t\ \gamma)$ and $d' = \mathsf{pj}_2\ (h\ s\ t\ \gamma)$, with $\mathsf{pj}_i : p_1 \wedge p_2 \to p_i\ (i = 1, 2)$.

However, it is often cumbersome to construct such bisimulations. The reason is that for a relation R to be a bisimulation, it needs (a) to be closed under

taking tail, (b) include all lemmas, and (c) all compositions of the required causal contexts. This typically gives rise to a large, or infinite relation.

We borrow a solution from process algebra [16,18], namely the method of *bisimulation-up-to*. A bisimulation-up-to is a relation which is included in a bisimulation, but which typically is not a bisimulation itself. A relation R is a bisimulation-up-to $\mathcal{U}$ if for every $s\ R\ t$ we have $\mathsf{head}\ s = \mathsf{head}\ t$ and $\mathsf{tail}\ s\ \mathcal{U}(R)\ \mathsf{tail}\ t$. For suitable $\mathcal{U}$, it is possible to prove that $\mathcal{U}(R)$ is a bisimulation whenever R is a bisimulation-up-to $\mathcal{U}$. Since R can be substantially smaller than the enclosing bisimulation $\mathcal{U}(R)$, this may save a lot of work, because less pairs have to be checked.

Definition 6. Let R, R' be relations on stream terms. Then R *progresses to* R' if for every $s\ R\ t$ we have $\mathsf{head}\ s = \mathsf{head}\ t$ and $\mathsf{tail}\ s\ R'\ \mathsf{tail}\ t$.

Definition 7. Let $S, T \subseteq T_{\mathfrak{S}} \times T_{\mathfrak{S}}$, C a stream context and σ a substitution. We define $C[S] = \{\langle C[s], C[t]\rangle \mid \langle s,t\rangle \in S\}$, and $S^\sigma = \{\langle s^\sigma, t^\sigma\rangle \mid \langle s,t\rangle \in S\}$. For $x \in \mathcal{X}_{\mathfrak{B}}$ we let $S^{\mathsf{gen}_{\mathfrak{B}}(x)} = \{\langle s,t\rangle \mid \langle s^{x\mapsto i}, t^{x\mapsto i}\rangle \in S \text{ for all } i \in \{0,1\}\}$, and for $x \in \mathcal{X}_{\mathfrak{S}}$, $S^{\mathsf{gen}_{\mathfrak{S}}(x)} = \{\langle s,t\rangle \mid \langle s^{x\mapsto y::z}, t^{x\mapsto y::z}\rangle \in S \text{ for some } y \in \mathcal{X}_{\mathfrak{B}},\ z \in \mathcal{X}_{\mathfrak{S}} \text{ fresh for } s,t\}$. We abbreviate $S^{\mathsf{gen}} = \bigcup_{x\in\mathcal{X}_{\mathfrak{B}}} S^{\mathsf{gen}_{\mathfrak{B}}(x)} \cup \bigcup_{x\in\mathcal{X}_{\mathfrak{S}}} S^{\mathsf{gen}_{\mathfrak{S}}(x)}$. Also, we let $S^{-1} = \{\langle s,t\rangle \mid \langle t,s\rangle \in S\}$ and $S \cdot T = \{\langle s,t\rangle \mid \langle s,u\rangle \in S, \langle u,t\rangle \in T\}$.

Definition 8. Let $R \subseteq T_{\mathfrak{S}} \times T_{\mathfrak{S}}$. We define $\mathcal{U}(R)$ inductively by the grammar

$$\mathcal{U}(R) ::= R \mid \sim \mid \mathcal{U}(R)^\sigma \mid C[\mathcal{U}(R)] \mid \mathcal{U}(R)^{-1} \mid \mathcal{U}(R)\cdot\mathcal{U}(R) \mid \mathcal{U}(R)^{\mathsf{gen}}$$

where C is a causal context. R is a *bisimulation-up-to* $\mathcal{U}$ if R progresses to $\mathcal{U}(R)$.

In words, $\mathcal{U}(R)$ is the smallest relation that contains R and $\sim$, and is closed under substitution, causal contexts, symmetry, transitivity and generalization. Actually, the clauses for substitution and generalization are immediate in Coq, as there the pairs in $\mathcal{U}(R)$ are explicitly universally quantified.

Theorem 9 (Soundness). *If R is a bisimulation-up-to $\mathcal{U}$, then $\mathcal{U}(R)$ is a bisimulation.*

Proof. Let R be a bisimulation-up-to $\mathcal{U}$, and let s,t be terms such that $s\ \mathcal{U}(R)\ t$. We prove $\mathsf{head}\ s = \mathsf{head}\ t$ and $\mathsf{tail}\ s\ \mathcal{U}(R)\ \mathsf{tail}\ t$ by induction on the definition of $s\ \mathcal{U}(R)\ t$. We distinguish the following cases:

(i) $s\ R\ t$: follows from R being a bisimulation-up-to $\mathcal{U}$;

(ii) $s \sim t$: $\mathsf{head}\ s = \mathsf{head}\ t$ and $\mathsf{tail}\ s \sim \mathsf{tail}\ t$, and so $\mathsf{tail}\ s\ \mathcal{U}(R)\ \mathsf{tail}\ t$;

(iii) $s\ \mathcal{U}(R)^\sigma\ t$: for some $u, v \in T_{\mathfrak{S}}$, we have $s = u^\sigma$, $t = v^\sigma$ and $u\ \mathcal{U}(R)\ v$. By the induction hypothesis (IH) we have $\mathsf{head}\ u = \mathsf{head}\ v$ and so $\mathsf{head}\ s = \mathsf{head}\ t$; also $\mathsf{tail}\ u\ \mathcal{U}(R)\ \mathsf{tail}\ v$ by IH and so $\mathsf{tail}\ s\ \mathcal{U}(R)^\sigma\ \mathsf{tail}\ t$;

(iv) $s\ C[\mathcal{U}(R)]\ t$: for some $u, v \in T_{\mathfrak{S}}$, $s = C[u]$, $t = C[v]$, and $u\ \mathcal{U}(R)\ v$. Then $\mathsf{head}\ s = \mathsf{head}\ t$ follows from IH and causality of C; moreover, $\mathsf{tail}\ s\ D[\mathcal{U}(R)]\ \mathsf{tail}\ t$ follows from causality of $D = \mathsf{tail}\ C[\mathsf{head}\ u :: \Box]$ and IH;

(v) $s\ \mathcal{U}(R)^{-1}\ t$: direct from IH;

(vi) $s\ \mathcal{U}(R)\ u\ \mathcal{U}(R)\ t$: direct from IH;

(vii) $s\ \mathcal{U}(R)^{\mathsf{gen}_{\mathfrak{B}}(x)}\ t$: then $s^{x\mapsto i}\ \mathcal{U}(R)\ t^{x\mapsto i}$ for $i \in \{0,1\}$ and so by IH $\mathsf{head}\ s = \mathsf{head}\ t$ for all possible values of $x \in \mathcal{X}_{\mathfrak{B}}$, and $\mathsf{tail}\ s\ \mathcal{U}(R)^{\mathsf{gen}_{\mathfrak{B}}(x)}\ \mathsf{tail}\ s$ by IH;

(viii) $s\ \mathcal{U}(R)^{\mathsf{gen}_{\mathfrak{S}}(x)}\ t$: similar to previous case.

From a proof using circular coinduction, we extract a bisimulation-up-to $\mathcal{U}$:

Definition 10. Let $d : \Gamma, \Delta \vdash s \sim^{\mathsf{cc}} t$ be a proof using circular coinduction. We define R_d to consist of all pairs (u, v) such that the proof d contains a sub-proof of the form $\sim^{\mathsf{cc}}_{\mathsf{coin}}\ \ldots : u \sim^{\mathsf{cc}} v$.

In what follows, we assume all contexts to be compatible with bisimilarity, i.e., $u \sim v \Rightarrow C[u] \sim C[v]$. This is used in item (i) of the proof of Theorem 12 below. In practice, this assumption can be dropped as proving it for concrete C forms no problem.

Lemma 11. *If R progresses to $\overline{R}$, and S progresses to $\overline{S}$, then $R \cup S$ progresses to $\overline{R} \cup \overline{S}$.* □

Theorem 12. *Assume $\Gamma, \Delta \vdash d : s \sim^{\mathsf{cc}} t$ and $u \sim v$ for all pairs $\gamma : u \sim^{\mathsf{cc}} v$ in Γ. Let $R = R_d \cup \Delta$. Then $s\ \mathcal{U}(R)\ t$ and R_d progresses to $\mathcal{U}(R)$.*

Proof. The proof proceeds by induction on the structure of $\Gamma, \Delta \vdash d : s \sim^{\mathsf{cc}} t$ (see Figure 1), as follows. In each case, we let R abbreviate $R_d \cup \Delta$. In the first three cases, we have $R_d = \varnothing$ and so R_d trivially progresses to $\mathcal{U}(R)$.

(i) $\Gamma, \Delta \vdash d : C[u^\sigma] \sim^{\mathsf{cc}} C[v^\sigma]$ with $d = \sim^{\mathsf{cc}}_{\mathsf{hyp}}\ \gamma\ C\ \sigma$. Then $(\gamma : u \sim^{\mathsf{cc}} v) \in \Gamma$, and $u^\sigma \sim v^\sigma$ by assumption. Hence, by compatibility we have $C[u^\sigma] \sim C[v^\sigma]$ and, using $\sim \subseteq \mathcal{U}(R)$ we get $C[u^\sigma]\ \mathcal{U}(R)\ C[v^\sigma]$.

(ii) $\Gamma, \Delta \vdash d : D[u^\sigma] \sim^{\mathsf{cc}} D[v^\sigma]$ with $d = \sim^{\mathsf{cc}}_{\mathsf{cohyp}}\ \delta\ D\ \sigma$, and D a causal context. Then $(\delta : u \sim^{\mathsf{cc}} v) \in \Delta$ and so $u\ R\ v$, $u^\sigma\ \mathcal{U}(R)\ v^\sigma$ and $D[u^\sigma]\ \mathcal{U}(R)\ D[v^\sigma]$.

(iii) $\Gamma, \Delta \vdash d : u \sim^{\mathsf{cc}} u$ with $d = \sim^{\mathsf{cc}}_{\mathsf{refl}}\ u$. Then $u\ \mathcal{U}(R)\ u$ by reflexivity of $\sim$ and $\sim \subseteq \mathcal{U}(R)$.

(iv) $\Gamma, \Delta \vdash d : u \sim v$ with $d = \sim^{\mathsf{cc}}_{\mathsf{sym}}\ d_1$ and $\Gamma, \Delta \vdash d_1 : v \sim u$. Then $v\ \mathcal{U}(R)\ u$ by the induction hypothesis (IH), and $u\ \mathcal{U}(R)\ v$ by symmetry of $\mathcal{U}(R)$. Also, R_d progresses to $\mathcal{U}(R)$ by IH, because $R_d = R_{d_1}$.

(v) $\Gamma, \Delta \vdash d : u \sim^{\mathsf{cc}} v$ with $d = \sim^{\mathsf{cc}}_{\mathsf{trans}}\ d_1\ d_2$, and $\Gamma, \Delta \vdash d_1 : u \sim^{\mathsf{cc}} w$ and $\Gamma, \Delta \vdash d_2 : w \sim^{\mathsf{cc}} v$ for some stream term w. We conclude $u\ \mathcal{U}(R)\ v$ from $u\ \mathcal{U}(R)\ w$, and $w\ \mathcal{U}(R)\ v$ by IH and transitivity of $\mathcal{U}(R)$. By IH we have that R_{d_1} progresses to $\mathcal{U}(R_{d_1} \cup \Delta)$, and R_{d_2} progresses to $\mathcal{U}(R_{d_2} \cup \Delta)$. By Lemma 11 $R_d = R_{d_1} \cup R_{d_2}$ progresses to $\mathcal{U}(R)$.

(vi) $\Gamma, \Delta \vdash d : u \sim^{\mathsf{cc}} v$ with $d = \sim^{\mathsf{cc}}_{\mathsf{coin}}\ \delta\ d_0\ d'$ and $\Gamma \vdash d_0 : \mathsf{head}\ u = \mathsf{head}\ v$, and $\Gamma, \Delta' \vdash d' : \mathsf{tail}\ u \sim^{\mathsf{cc}} \mathsf{tail}\ v$ where $\Delta' = \Delta \cup \{\delta : u \sim^{\mathsf{cc}} v\}$. Note that $R_d = R_{d'} \cup \{\delta : u \sim^{\mathsf{cc}} v\}$ and so $R = R_d \cup \Delta = R_{d'} \cup \Delta'$. From the IH we obtain that $R_{d'}$ progresses to $\mathcal{U}(R_{d'} \cup \Delta') = \mathcal{U}(R)$. Moreover, $\{\delta : u \sim^{\mathsf{cc}} v\}$ progresses to $\mathcal{U}(R)$ since d_0 is a proof of $\mathsf{head}\ u = \mathsf{head}\ v$, and $\mathsf{tail}\ u\ \mathcal{U}(R)\ \mathsf{tail}\ v$ by IH. With Lemma 11 we conclude that R_d progresses to $\mathcal{U}(R)$. Furthermore, $u\ \mathcal{U}(R)\ v$ by $\langle u, v\rangle \in R_d \subseteq R \subseteq \mathcal{U}(R)$.

(vii) $\Gamma, \Delta \vdash d : u \sim^{\mathsf{cc}} v$ with $d = \sim^{\mathsf{cc}}_{\mathsf{cut}}\ \gamma\ d_1\ d_2$ and $\Gamma', \Delta \vdash d_1 : u \sim^{\mathsf{cc}} v$, and $\Gamma, \varnothing \vdash d_2 : q \sim^{\mathsf{cc}} r$ for some stream terms q, r, where $\Gamma' = \Gamma \cup \{\gamma : q \sim^{\mathsf{cc}} r\}$. We note that $q \sim r$ holds by Theorem 9, since by IH we have $q\ \mathcal{U}(R_{d_2})\ r$, and R_{d_2} progresses to $\mathcal{U}(R_{d_2})$. Hence $\Gamma' \subseteq \sim$, and by IH we obtain $u\ \mathcal{U}(R_{d_1} \cup \Delta)\ v$ and we conclude $u\ \mathcal{U}(R)\ v$ from $R_d = R_{d_1} \cup R_{d_2}$ (clearly, $\mathcal{U}$ is a monotone function with respect to $\subseteq$). Moreover, by IH R_{d_1} progresses to $\mathcal{U}(R_{d_1} \cup \Delta)$, and R_{d_2} progresses to $\mathcal{U}(R_{d_2})$. So by Lemma 11 we obtain that $R_d = R_{d_1} \cup R_{d_2}$ progresses to $\mathcal{U}(R)$.

(viii) $\Gamma, \Delta \vdash d : u \sim^{\mathsf{cc}} v$ with $d = \sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{B}}}\ x\ d_0\ d_1$ and $\Gamma, \Delta \vdash d_i : (u \sim^{\mathsf{cc}} v)^{x \mapsto i}$ for $i \in \{0, 1\}$ and $x \in \mathcal{X}_{\mathfrak{B}}$. By IH we get $u^{x \mapsto i}\ \mathcal{U}(R)\ v^{x \mapsto i}$ for $i \in \{0, 1\}$, hence $u\ \mathcal{U}(R)\ v$ by $\mathcal{U}(R)^{\mathsf{gen}_{\mathfrak{B}}} \subseteq \mathcal{U}(R)$. Furthermore, $R_d = R_{d_0} \cup R_{d_1}$ progresses to $\mathcal{U}(R)$ by IH and Lemma 11.

(ix) $\Gamma, \Delta \vdash d : u \sim^{\mathsf{cc}} v$ with $d = \sim^{\mathsf{cc}}_{\mathsf{case}_{\mathfrak{S}}}\ x\ e$ and $\Gamma, \Delta \vdash e : u^{x \mapsto y::z} \sim^{\mathsf{cc}} v^{x \mapsto y::z}$. From IH we obtain $u^{x \mapsto y::z}\ \mathcal{U}(R)\ v^{x \mapsto y::z}$. So $u\ \mathcal{U}(R)\ v$ follows from $\mathcal{U}(R)^{\mathsf{gen}_{\mathfrak{S}}} \subseteq \mathcal{U}(R)$. Furthermore, $R_d = R_e$ progresses to $\mathcal{U}(R)$ by IH.

From Theorems 9 and 12 it follows that every proof by circular coinduction can be transformed to a bisimilarity proof accepted by Coq. We have formalized Theorem 9 in Coq, see [10]. We are currently working on automating the translation of circular coinductive proofs as produced by the prover Circ [12,15], or Streambox [21] into Coq proofs.

Corollary 13. *If $\Gamma, \varnothing \vdash d : s \sim^{\mathsf{cc}} t$, then R_d is a bisimulation-up-to $\mathcal{U}$ and $s\ \mathcal{U}(R_d)\ t$. Hence $s \sim t$ is provable from Γ in Coq.* □

6 Discussion

Generally speaking, for *programming* with infinite objects, translating a productive specification into a guarded definition may take considerable effort. In doing so, the elegance and 'directness' of the original specification is often lost, thereby complicating further processing of the defined object. For programming, we therefore believe that alternative approaches are favorable. Here one may think of implementing a more flexible productivity checker by using a type-based approach as advocated in [13,1,20]. It would also be convenient if Coq would allow productivity of the corecursive program to be proved separately (for recursive programs, wellfoundedness can be proved separately in Coq). This then would open the door for more advanced tactics based on methods as described in [7,8,9].

However, for *reasoning* about infinite objects the situation is different, as we have shown. The reason is that for bisimilarity proofs the coinduction hypothesis is usually not subject to further pattern matching. This is in contrast to programming where recursive calls are usually manipulated further.

References

1. Abel, A.: Termination and Productivity Checking with Continuous Types. In: Hofmann, M.O. (ed.) TLCA 2003. LNCS, vol. 2701, pp. 1–15. Springer, Heidelberg (2003)

2. The Agda team. The Agda Wiki (2011), http://wiki.portal.chalmers.se/agda
3. Allouche, J.-P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press, New York (2003)
4. The Coq development team. The Coq Proof Assistant Reference Manual. LogiCal Project, version 8.3 (2012), http://coq.inria.fr
5. Coquand, T.: Infinite Objects in Type Theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)
6. Danielsson, N.A.: Beating the Productivity Checker Using Embedded Languages. In: Proc. Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010). EPTCS, vol. 43, pp. 29–48 (2010)
7. Endrullis, J., Grabmayer, C., Hendriks, D.: Data-Oblivious Stream Productivity. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 79–96. Springer, Heidelberg (2008)
8. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.W.: Productivity of Stream Definitions. Theoretical Computer Science 411 (2010)
9. Endrullis, J., Hendriks, D.: Lazy Productivity via Termination. Theoretical Computer Science 412(28), 3203–3225 (2011)
10. Endrullis, J., Hendriks, D., Bodin, M.: Coq Formalization for Circular Coinduction (2012), http://www.cs.vu.nl/~diem/research/up_to.tgz
11. Giménez, E.: Codifying Guarded Definitions with Recursive Schemes. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 39–59. Springer, Heidelberg (1994)
12. Goguen, J., Lin, K., Roşu, G.: Circular Coinductive Rewriting. In: Proc. of Automated Software Engineering, pp. 123–131. IEEE (2000)
13. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: Symposium on Principles of Programming Languages (POPL 1996), pp. 410–423 (1996)
14. Hur, C.-K., Neis, G., Dreyer, D., Vafeiadis, V.: The Power of Parameterization in Coinductive Proof. In: Proc. Symp. on Principles of Programming Languages (POPL 2013), pp. 193–206. ACM (2013)
15. Lucanu, D., Roşu, G.: Circular Coinduction with Special Contexts. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 639–659. Springer, Heidelberg (2009)
16. Milner, R.: Communication and Concurrency. Prentice-Hall International Series in Computer Science. Prentice-Hall (1989)
17. Post, E.L.: Formal Reductions of the General Combinatorial Decision Problem. American Journal of Mathematics (65), 197–215 (1943)
18. Pous, D., Sangiorgi, D.: Enhancements of the Coinductive Proof Method. In: Sangiorgi, D., Rutten, J.J.M.M. (eds.) Advanced Topics in Bisimulation and Coinduction. Cambridge Tracts in Theoretical Computer Science, vol. 52, ch. 6, Cambridge University Press, Cambridge (2011)
19. Roşu, G., Lucanu, D.: Circular Coinduction: A Proof Theoretical Foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 127–144. Springer, Heidelberg (2009)
20. Severi, P., de Vries, F.-J.: Pure Type Systems with Corecursion on Streams: From Finite to Infinitary Normalisation. In: Proc. Int. Conf. on Functional Programming (ICFP 2012), pp. 141–152. ACM (2012)
21. Zantema, H., Endrullis, J.: Proving Equality of Streams Automatically. In: Proc. Conf. on Rewriting Techniques and Applications (RTA 2011). LIPIcs, vol. 10, pp. 393–408. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011)

Program Extraction from Nested Definitions

Kenji Miyamoto[1,⋆], Fredrik Nordvall Forsberg[2,⋆,⋆⋆]
and Helmut Schwichtenberg[1]

[1] Ludwig-Maximilians-Universität München, Germany
[2] Swansea University, UK

Abstract. Minlog is a proof assistant which automatically extracts computational content in an extension of Gödel's T from formalized proofs. We report on extending Minlog to deal with predicates defined using a particular combination of induction and coinduction, via so-called nested definitions. In order to increase the efficiency of the extracted programs, we have also implemented a feature to translate terms into Haskell programs. To illustrate our theory and implementation, a formalisation of a theory of uniformly continuous functions due to Berger is presented.

1 Introduction

Program extraction is a method for obtaining certified algorithms by extracting the computational content hidden in proofs. To get successful algorithms, the formalization of the proof is not a superficial issue but rather an essential one. The *Theory of Computable Functionals* [24], TCF in short, has been developed in order to provide a concrete framework for program extraction. TCF is implemented straightforwardly in the *Minlog* [18] proof assistant. As available in TCF, Minlog supports inductive and coinductive definitions and program extraction from classical proofs as well as from constructive ones. The internal term language of Minlog can be exported to general-purpose programming languages.

This paper reports on new contributions to TCF and Minlog, focusing on two aspects. One is a certain combination of inductive and coinductive definitions, called *nested definitions* [6]. We make use of such definitions in a case study on exact real arithmetic. The other is a feature to translate Minlog algebras and terms into Haskell programs. This makes efficient execution of the extracted programs possible. Translation to a lazy language such as Haskell is especially beneficial when computing with infinite objects, such as in our case study.

We first describe TCF, with an emphasis on nested definitions. Then an application of TCF and program extraction from nested definitions to exact real arithmetic is presented: a translation of the usual type-1 representation of uniformly continuous functions into a type-0 representation. We also extract a program which computes the definite integral of such functions. These case studies are available in the Minlog distribution [18].

⋆ The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement number 238381.
⋆⋆ Supported by EPSRC grant EP/G033374/1.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 370–385, 2013.

2 Formal System

We study higher type functionals as well as functions and ground type objects. Functionals in TCF are not necessarily total but partial in general. Based on the understanding that evaluation must be finite, we assume two principles for our notion of computability: the finite support principle and the monotonicity principle. During the evaluation of some functional Φ, only finitely many inputs $\varphi_0, \ldots, \varphi_{n-1}$ are used. Moreover, each of φ_i must be presented to Φ in a finite form. This is the finite support principle. Assume $\Phi(\varphi_0)$ evaluates to a value k and let φ_1 be more informative than φ_0. Then $\Phi(\varphi_1)$ results in k as well. This is the monotonicity principle.

The notion of abstract computability is formulated as follows: an object is computable when its set of finite approximations is primitive recursively enumerable. In this section, we begin by making the notion of such computable objects, called partial continuous functionals, concrete. Then we proceed to our term calculus, its Haskell translation, and inductive/coinductive predicates.

2.1 Algebras and Their Total and Cototal Ideals

The formal term language of TCF is an extension of Gödel's T, which is appropriate for higher type computation involving functionals. Types are built from base types by the formation of function types. The base types themselves are formed by free algebras given by their constructors. For instance the list algebra $\mathbf{L}_\alpha$, where α is a type parameter, is defined by the two constructors empty list $[]^{\mathbf{L}_\alpha}$ and the "cons" operator $::^{\alpha\to\mathbf{L}_\alpha\to\mathbf{L}_\alpha}$. Formally, a constructor type is a type expression of the form $\tau_0 \to \ldots \to \tau_{n-1} \to \xi_i$, where each τ_i is a type expression where all $\vec{\xi}$ appear strictly positively (i.e. not to the left of an arrow). For any finite list of constructor types $\vec{\kappa}$, we (simultaneously) define algebras $\mu_{\vec{\xi}}(\vec{\kappa})$, provided there is at least one constructor type such that $\vec{\xi}$ does not occur in $\vec{\tau}$ (this ensures that all algebras are inhabited). For example, the algebra of natural numbers $\mathbf{N}$ is defined by $\mathbf{N} = \mu_\xi(\xi, \xi \to \xi)$. We also adopt the notation $\mathbf{L}_\alpha = \mu_\xi([]^\xi, ::^{\alpha\to\xi\to\xi})$ in order to specify constructor names. Another example is the algebra of branching trees. We simultaneously define $(\mathbf{Ts}, \mathbf{T})$ by $\mu_{\xi,\zeta}(\mathsf{Empty}^\xi, \mathsf{Tcons}^{\zeta\to\xi\to\xi}, \mathsf{Leaf}^\zeta, \mathsf{Branch}^{\xi\to\zeta})$. Using the list algebra, we can define another algebra of branching trees without the simultaneity by defining $\mathbf{NT} = \mu_\xi(\mathsf{Lf}^\xi, \mathsf{Br}^{\mathbf{L}_\xi\to\xi})$. This is an example of a *nested* algebra [6]. Support for such algebras has recently been added to Minlog.

The intended semantics of the term language is based on Scott's information systems [25] (see also Schwichtenberg and Wainer [24]). Algebras are interpreted as sets of *ideals*, i.e. consistent and deductively closed sets of tokens, which are type correct constructor trees possibly involving the special symbol $*$, meaning "no information". Consider a constructor tree $P(*)$ with a distinguished occurrence of $*$. An arbitrary $P(C\vec{*})$, where C is a constructor, is called a *one-step predecessor* of $P(*)$, written $P(C\vec{*}) \succ_1 P(*)$. Here $P(C\vec{*})$ is obtained by substituting $C\vec{*}$ for the distinguished $*$ in $P(*)$. Among ideals, we are especially

interested in total and cototal ideals. A *cototal ideal* x is an ideal whose every constructor tree $P(*) \in x$ has a one-step predecessor $P(C\vec{*}) \in x$. A *total ideal* is a cototal ideal such that the relation $\succ_1$ is well founded. For instance, the cototal ideal $\{*::*, 0::*, *::*::*, 0::*::*, *::1::*, 0::1::*, *::*::*::*, \ldots\}$ denotes the non-well founded list of natural numbers $[0, 1, \ldots]$.

A binary tree with a (possibly) infinite height is informally defined by a term $t := \mathsf{Br}(t::t::[])$ whose denotation is an $\mathbf{NT}$-cototal $\mathbf{L_{NT}}$-total ideal (see also Section 2.5). Total ideals of $(\mathbf{Ts}, \mathbf{T})$ are isomorphic (as information systems) to pairs of $\mathbf{L_{NT}}$-total $\mathbf{NT}$-total ideals and $\mathbf{NT}$-total $\mathbf{L_{NT}}$-total ideals.

2.2 Corecursion

An arbitrary term in Gödel's T is terminating and hence denotes a total ideal. Constructors are used to construct total ideals whereas recursion operators are used to inspect a total ideal from its leaves to the root. In order to accommodate cototal as well as total ideals, we add to Gödel's T two more kinds of constants, namely destructors and corecursion operators, which this section describes. Destructors, the dual of constructors, are used to inspect the structure of cototal ideals, while corecursion operators give a way to construct cototal ideals.

As an example, we consider the algebra $\mathbf{NT}$ of nested trees. Define the disjoint sum of α and β by $\alpha + \beta = \mu_\xi(\mathsf{inl}^{\alpha\to\xi}, \mathsf{inr}^{\beta\to\xi})$, and the unit type $\mathbf{U} = \mu_\xi(\mathsf{u}^\xi)$. The destructor $\mathcal{D}_{\mathbf{NT}}$ has the following type and conversion relation:

$$\begin{aligned} &\mathcal{D}_{\mathbf{NT}} : \mathbf{NT} \to \mathbf{U} + \mathbf{L_{NT}} \\ &\mathcal{D}_{\mathbf{NT}}\,\mathsf{Lf} \mapsto \mathsf{inl}\ \mathsf{u}, \qquad \mathcal{D}_{\mathbf{NT}}\,(\mathsf{Br}\ as) \mapsto \mathsf{inr}\ as. \end{aligned} \tag{1}$$

Corecursion operators give a way to construct cototal ideals. The corecursion operator ${}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}}$ has the following type and conversion relation:

$$\begin{aligned} &{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}} : \tau \to (\tau \to \mathbf{U} + \mathbf{L}_{\mathbf{NT}+\tau}) \to \mathbf{NT} \\ &{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}}\,N\,M \mapsto \text{case } MN \text{ of} \\ &\qquad\qquad \mathsf{inl}\ \mathsf{u} \to \mathsf{Lf} \\ &\qquad\qquad \mathsf{inr}\ as \to \mathsf{Br}\ (\mathcal{M}^{\mathbf{NT}+\tau\to\mathbf{NT}}_{\lambda_\alpha \mathbf{L}_\alpha}\ as\ [\mathrm{id}, \lambda_x({}^{\mathrm{co}}\mathcal{R}xM)]). \end{aligned}$$

where $[f, g]^{\rho+\sigma\to\tau}$ is defined for $f^{\rho\to\tau}$ and $g^{\sigma\to\tau}$ by

$$[f, g](\mathsf{inl}\ x^\rho) \mapsto fx, \qquad [f, g](\mathsf{inr}\ y^\sigma) \mapsto gy,$$

and the map constant $\mathcal{M}^{\sigma\to\rho}_{\lambda_\alpha \mathbf{L}_\alpha}$ witnesses the functoriality of $\mathbf{L}_\alpha$. It has the following type and conversion relation:

$$\begin{aligned} &\mathcal{M} : \mathbf{L}_\sigma \to (\sigma \to \rho) \to \mathbf{L}_\rho \\ &\mathcal{M}\ []\ f \mapsto [], \qquad \mathcal{M}\ (x :: xs)\ f \mapsto fx :: (\mathcal{M}\ xs\ f). \end{aligned}$$

In the conversion rule for ${}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{NT}}$, the first argument of the corecursion operator is passed to the second functional argument. The result of this application determines what the construction of the cototal ideal is. In the case of nested algebras,

the value algebra of corecursion operators occurs inside of other algebras as a parameter. Map operators play a crucial role in reaching the value algebra so that the corecursion operator can do its work.

2.3 Realizability

We now address the issue of extracting computational content from proofs. The method of program extraction is based on *modified realizability* as introduced by Kreisel [16] and described in detail in Schwichtenberg and Wainer [24]. In short, from every constructive proof M of a non-Harrop formula A (in natural deduction) one extracts a program $\mathrm{et}(M)$ "realizing" A, essentially by removing computationally irrelevant parts from the proof (proofs of Harrop formulas have no computational content). The extracted program has some simple type $\tau(A)$ which depends solely on the logical shape of the proven formula A. In its original form the extraction process is fairly straightforward, but often leads to unnecessarily complex programs. In order to obtain better programs, proof assistants (for instance Coq [9], Isabelle/HOL [13], Agda [1], Nuprl [21], Minlog [18]) offer various optimizations of program extraction. Below we describe optimizations implemented in Minlog [22], which are relevant for our present case study.

Quantifiers without Computational Content. Besides the usual quantifiers, $\forall$ and $\exists$, Minlog has so-called *non-computational quantifiers*, $\forall^{\mathrm{nc}}$ and $\exists^{\mathrm{nc}}$, which allow for the extraction of simpler programs. These quantifiers, which were first introduced by Berger [2], can be viewed as a refinement of the Set/Prop distinction in constructive type systems like Coq. Intuitively, a proof of $\forall^{\mathrm{nc}}_x A(x)$ ($A(x)$ non-Harrop) represents a procedure that assigns to any x a proof $M(x)$ of $A(x)$ where $M(x)$ does not make "computational use" of x, i.e., the extracted program $\mathrm{et}(M(x))$ does not depend on x. Dually, a proof of $\exists^{\mathrm{nc}}_x A(x)$ is a proof of $M(x)$ for some x where the witness x is "hidden", that is, not available for computational use; in fact, $\exists^{\mathrm{nc}}$ can be seen as inductively defined by the clause $\forall^{\mathrm{nc}}_x(A \to \exists^{\mathrm{nc}}_x A)$. The types of extracted programs for non-computational quantifiers are $\tau(\forall^{\mathrm{nc}}_{x^\rho} A) = \tau(\exists^{\mathrm{nc}}_{x^\rho} A) = \tau(A)$ as opposed to $\tau(\forall_{x^\rho} A) = \rho \to \tau(A)$ and $\tau(\exists_{x^\rho} A) = \rho \times \tau(A)$. The extraction rules are, for example in the case of $\forall^{\mathrm{nc}}$-introduction and -elimination, $\mathrm{et}((\lambda_x M^{A(x)})^{\forall^{\mathrm{nc}}_x A(x)}) = \mathrm{et}(M)$ and $\mathrm{et}((M^{\forall^{\mathrm{nc}}_x A(x)} t)^{A(t)}) = \mathrm{et}(M)$ as opposed to $\mathrm{et}((\lambda_x M^{A(x)})^{\forall_x A(x)}) = \mathrm{et}(\lambda_x M)$ and $\mathrm{et}((M^{\forall_x A(x)} t)^{A(t)}) = \mathrm{et}(Mt)$. For the extracted programs to be correct the variable condition for $\forall^{\mathrm{nc}}$-introduction must be strengthened by additionally requiring that the abstracted variable x does not occur in the extracted program $\mathrm{et}(M)$, and similarly for $\exists^{\mathrm{nc}}$. Note that for a Harrop formula A the formulas $\forall^{\mathrm{nc}}_x A$, $\forall_x A$ are equivalent.

2.4 Translation to a General-Purpose Programming Language

The programs extracted from proofs in Minlog are once again represented as terms in the internal term language. This has the advantage that a general

soundness theorem can be stated (and automatically proven) in the system. However, for efficiency and interoperability reasons, it is sometimes beneficial to translate the extracted terms into programs in a general-purpose programming language. We now describe our new translation from Minlog terms into Haskell programs (there is also limited support for translating into Scheme). Coq [17], Isabelle/HOL [5] and Agda [26] provide similar features.

Terms of Gödel's T – lambda abstraction, application, variables etc – are translated to corresponding Haskell terms. For recursion and corecursion operators, polymorphic functions are generated. For example, the translation of the corecursion operator ${}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{NT}}$ above is implemented as

```
ntCoRec :: a -> (a -> Maybe [Either NT a]) -> NT
ntCoRec e f = case (f e) of
                Nothing -> Lf
                Just z  ->
                  Br (fmap (\ w -> case w of
                                     Left x -> x
                                     Right y -> ntCoRec y f) z)
```

Here Haskell's lazy evaluation means that we do not need to worry about guarding the recursive call. The occurrence of the map operator $\mathcal{M}^{\mathbf{NT}+\tau\to\mathbf{NT}}_{\lambda_\alpha \mathbf{L}_\alpha}$ gets translated to the `fmap` function from the `Functor` type class. In this case, the list algebra from Minlog gets translated to the list data type in Haskell, which already has a `Functor` instance. For custom data types, the instance is derived automatically by GHC using the `DeriveFunctor` flag.

Lists, integers, rational numbers, sum types, product types and the unit type are translated to their standard implementation in the Haskell prelude. For efficiency reasons, natural numbers are translated to integers. Other algebras are translated into algebraic data types.

Program constants and their computation rules are translated to functions defined by pattern matching. Here some care must be taken for e.g. natural numbers, since they are translated to integers, for which no pattern matching is available. Instead *guard conditions* are used, as in the translation of the following parity function for natural numbers:

```
parity :: Integer {-Nat-} -> Bool
parity 0 = True
parity 1 = False
parity n | n > 1 = parity (n - 2)
```

The realizer for *ex-falso-quodlibet* $\mathsf{ff} \to A$ makes use of a *canonical inhabitant* $\mathsf{inhab}_{\tau(A)}$ of type $\tau(A)$. This is justified since all types are inhabited in the intended semantics, but not so in Haskell. Hence we define a type class

```
class Inhabited a where
  inhab :: a
```

and ensure we generate instances and track inhabitedness constraints in the types of the generated functions.

2.5 Inductive and Coinductive Definitions

We are particularly interested in dealing with a combination of induction and coinduction in TCF. Starting from simultaneous inductive definitions, we describe nested inductive definitions and furthermore nested inductive/coinductive definitions. See e.g. Jacobs and Rutten [14] for a gentle introduction to coinduction.

As an example, consider the simultaneously defined algebras $(\mathbf{Ts}, \mathbf{T}) = \mu_{\xi,\zeta}(\mathsf{Empty}^{\xi}, \mathsf{Tcons}^{\zeta\to\xi\to\xi}, \mathsf{Leaf}^{\zeta}, \mathsf{Branch}^{\xi\to\zeta})$ of finitely branching trees again. The totality predicates $(T_{\mathbf{Ts}}, T_{\mathbf{T}})$ of $(\mathbf{Ts}, \mathbf{T})$, of arity $(\mathbf{Ts})$ and $(\mathbf{T})$ respectively, are simultaneously inductively defined by the following four clauses:

$$T_{\mathbf{Ts}}\ \mathsf{Empty}, \qquad \forall^{\mathrm{nc}}_{a,as}(T_{\mathbf{T}}\ a \to T_{\mathbf{Ts}}\ as \to T_{\mathbf{Ts}}\ (\mathsf{Tcons}\ a\ as)),$$
$$T_{\mathbf{T}}\ \mathsf{Leaf}, \qquad \forall^{\mathrm{nc}}_{as}(T_{\mathbf{Ts}}\ as \to T_{\mathbf{T}}\ (\mathsf{Branch}\ as)).$$

From the above, a nested definition of branching trees is derived by removing the simultaneity. This leads to the definition of the algebras $\mathbf{L}_\alpha = \mu_\xi([]^{\xi}, ::^{\alpha\to\xi\to\xi})$ and $\mathbf{NT} = \mu_\xi(\mathsf{Lf}^{\xi}, \mathsf{Br}^{\mathbf{L}_\xi\to\xi})$. To define the totality predicate for $\mathbf{NT}$, we first define the relativised totality predicate RT_X for lists, with arity $(\mathbf{L}_\alpha)$ for X of arity (α). Relativised totality means the totality relative to the parameter predicate X. It is given by the following clauses:

$$RT_X\ [], \qquad \forall^{\mathrm{nc}}_{x,xs}(Xx \to RT_X\ xs \to RT_X\ (x::xs)).$$

We can now define the totality predicate of nested trees using the relativised totality predicate RT_X of lists, with X instantiated to $T_{\mathbf{NT}}$:

$$T_{\mathbf{NT}}\ \mathsf{Lf}, \qquad \forall_{as}(RT_{T_{\mathbf{NT}}}\ as \to T_{\mathbf{NT}}\ (\mathsf{Br}\ as)).$$

We call a predicate definition *nested* if the predicate to be defined occurs strictly positively as a parameter of an already defined predicate in a clause formula. Witnesses of nested predicates have nested algebras as their types.

Coinductive predicates arise as "duals" of inductive ones. For example, for the totality predicate $T_{\mathbf{NT}}$ we can define its companion predicate ${}^{\mathrm{co}}T_{\mathbf{NT}}$ by the single clause

$$\forall^{\mathrm{nc}}_{a}({}^{\mathrm{co}}T_{\mathbf{NT}}a \to a = \mathsf{Lf} \vee \exists^{\mathrm{nc}}_{as}(RT_{{}^{\mathrm{co}}T_{\mathbf{NT}}}as \wedge a = \mathsf{Br}as)). \tag{2}$$

We call such a companion predicate definition derived from an inductive one a *coinductive definition*. A witness for a proposition ${}^{\mathrm{co}}T_{\mathbf{NT}}\ a$ is an $\mathbf{NT}$-cototal $\mathbf{L}_{\mathbf{NT}}$-total ideal, which is a finitely branching tree of (possibly) infinite height. The computational content of (2) is the destructor $\mathcal{D}_{\mathbf{NT}}$. We still need to express that RT_X is the least predicate satisfying the clauses, and that ${}^{\mathrm{co}}T_{\mathbf{NT}}$ is the greatest predicate satisfying the clause. The former is done by means of the least-fixed-point axiom

$$\begin{aligned}\forall^{\mathrm{nc}}_{xs}(RT_X\ xs \to P\ [] \to\ & \\ & \forall^{\mathrm{nc}}_{x,xs}(Xx \to RT_X\ xs \to P\ xs \to P\ x::xs) \to \\ & P\ xs).\end{aligned} \tag{3}$$

The latter is done by means of the greatest-fixed-point axiom.

$$\forall_a^{\mathrm{nc}}(Q\ a \to \forall_a^{\mathrm{nc}}(Q\ a \to a = \mathsf{Lf} \vee \exists_{as}^{\mathrm{nc}}(RT_{{}^{\mathrm{co}}T_{\mathbf{NT}} \vee Q}\ as \wedge a = \mathsf{Br}\ as)) \to {}^{\mathrm{co}}T_{\mathbf{NT}}\ a). \tag{4}$$

The predicates P and Q are called competitor predicates which satisfy the same clause(s) as RT_X and $T_{\mathbf{NT}}$, respectively. From (3) and (4), we see that P is a superset of RT_X and Q is a subset of ${}^{\mathrm{co}}T_{\mathbf{NT}}$.

The term extracted from (3) is Gödel's (structural) recursion operator $\mathcal{R}_{\mathbf{L}_\alpha}$, and the term extracted from (4) is the corecursion operator ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{NT}}$ defined in Section 2.2.

3 Case Study: Uniformly Continuous Functions

To illustrate program extraction in TCF, we formalize the theory of uniformly continuous functions from constructive analysis [7,23]. Our first case study provides an alternative view of uniformly continuous functions of type-1 as a cototal object of type-0; i.e. of ground type. We then extract a program which computes the definite integral of a uniformly continuous function of type-0. This was first studied by Berger [3] in the setting of program extraction. We now offer machine extraction from formalized proofs of these results in Minlog. Before continuing, we review representations of real numbers of type-1 and type-0. In this section, we only consider real numbers and uniformly continuous functions in the interval $[-1, 1]$ in order to work with stream represented real numbers [8] and uniformly continuous functions on them.

A real number of type-1 is a Cauchy real with a modulus, namely a pair $\langle x, M\rangle$, where x is a bounded function of type $\mathbf{N} \to \mathbf{Q}$ and $M : \mathbf{N} \to \mathbf{N}$ satisfies the following Cauchy condition:

$$\forall_{k^{\mathbf{N}}} \forall_{n,m \geq Mk}(|x\,n - x\,m| \leq 2^{-k}).$$

Define the type of signed digits by $\mathbf{SD} = \mu_\xi(-1^\xi, 0^\xi, 1^\xi)$. A real number of type-0 is a signed digit stream $d_0{::}d_1{::}\ldots$, where d_i is of type $\mathbf{SD}$. Informally, the stream $d_0{::}d_1{::}\ldots$ denotes the real number $\sum_{i=0} \frac{d_i}{2^{i+1}}$. We represent such an object by a cototal ideal of $\mathbf{L}_{\mathbf{SD}}$, which is a possibly infinite list of signed digits. An arbitrary real number can be represented by a type-0 object, for example by a stream of integers in $\{-9, -8, \ldots, 8, 9\}$ with a decimal point [27,28].

3.1 Data Types of Uniformly Continuous Functions

Consider a triple $\langle h, \alpha, \omega\rangle$, where $h : \mathbf{Q} \to \mathbf{N} \to \mathbf{Q}$ is a bounded function and α and ω are of type $\mathbf{N} \to \mathbf{N}$. Suppose that it satisfies

$$\forall_{a^{\mathbf{Q}}, k^{\mathbf{N}}, n \geq \alpha(k), m \geq \alpha(k)}(|h\,a\,n - h\,a\,m| \leq 2^{-k}),$$
$$\forall_{a^{\mathbf{Q}}, b^{\mathbf{Q}}, k^{\mathbf{N}}, n \geq \alpha(k)}(|a - b| \leq 2^{-\omega(k)+1} \to |h\,a\,n - h\,b\,n| \leq 2^{-k}).$$

The first formula states the Cauchyness of $\langle h\, a, \alpha\rangle$. Classically this can be stated by the formula $\forall_{a,k}\exists_l\forall_{n,m\geq l}(|h\, a\, n - h\, a\, m| \leq 2^{-k})$, while constructively the way to determine l has to be given, for instance by a Cauchy modulus α. The second formula states the uniform continuity of $\langle h, \alpha\rangle$, once again with explicit modulus of uniform continuity ω for constructive reasons. Finally, we assume that h is bounded between -1 to 1. We adopt objects of this kind as our uniformly continuous functions of type-1, namely of first order function type. Application of a type-1 uniformly continuous function $\langle h, \alpha, \omega\rangle$ to a Cauchy real $\langle x, M\rangle$ is defined to be

$$\langle \lambda_n(h\,(x\, n)\, n), \lambda_k \max(\alpha(k+2), M(\omega(k+1)-1))\rangle.$$

For our type-0 representation of uniformly continuous functions we adopt so-called read-write machines [3] or stream processors [11,12]. These are $\mathbf{W}$-cototal $\mathbf{R}_\mathbf{W}$-total ideals where

$$\mathbf{R}_\alpha \coloneqq \mu_\xi(\mathsf{Put}^{\mathbf{SD}\to\alpha\to\xi}, \mathsf{Get}^{\xi\to\xi\to\xi\to\xi}),$$
$$\mathbf{W} \coloneqq \mu_\xi(\mathsf{Stop}^\xi, \mathsf{Cont}^{\mathbf{R}_\xi\to\xi}).$$

A read-write machine is a potentially non-well founded tree with internal Put nodes and branching at Get nodes. It intuitively represents a function from signed digit streams to signed digit streams as follows: start at the root of the tree. If we are at the node $(\mathsf{Put}\ d\ t)$, output the digit d and carry on with the tree t. If we are at the node $(\mathsf{Get}\ t_{-1}\ t_0\ t_1)$, read a digit d from the input stream and continue with the tree t_d. If we reach a Stop node, we return the rest of the input unprocessed as output. Because a read-write machine is a $\mathbf{W}$-cototal $\mathbf{R}_\mathbf{W}$-total ideal, the output might be infinite, but $\mathbf{R}_\mathbf{W}$-totality ensures that the machine can only read finitely many input digits before producing another output digit; the machine represents a continuous function.

3.2 Formalization

We work with the abstract theory of uniformly continuous functions. Suppose that φ is a type variable representing abstract uniformly continuous functions. Due to the use of non-computational connectives, any object of type φ appearing in the proofs will disappear when a program is extracted. This theory is axiomatized in Appendix A. Note that all axioms are non-computational.

Let f range over the type variable φ, and also p, q range over $\mathbf{Q}$ and k, l range over $\mathbf{N}$. We define a comprehension term C (for "continuous") of abstract uniformly continuous functions as follows.

$$\mathrm{C} \coloneqq \{f|\forall_k\exists_l \mathrm{B}_{l,k} f\},\ \text{where}\ \mathrm{B}_{l,k} \coloneqq \{f|\forall_p\exists_q(f[I_{p,l}] \subseteq I_{q,k})\}.$$

Here, $I_{q,k}$ represents the interval $[q - 2^{-k}, q + 2^{-k}]$ of length 2^{1-k} centered at q, while $f[I_{p,l}]$ represents the image of $I_{p,l}$ under f; the exact behavior is axiomatized in Appendix A. We write I for the interval $I_{0,0} = [-1, 1]$. Witnesses

for $\mathrm{B}_{l,k}f$ and $\mathrm{C}f$ are total ideals of type (an isomorphic copy of) $\mathbf{Q} \to \mathbf{Q}$ and $\mathbf{N} \to \mathbf{N} \times (\mathbf{Q} \to \mathbf{Q})$, respectively. The latter represents $\langle h, \alpha, \omega \rangle$ by a term $\lambda_n \langle \omega\, n, \lambda_a\, h\, a\, (\alpha\, n) \rangle$. Let $(\mathrm{Out}_d \circ f)(x) = 2f(x) - d$, $(f \circ \mathrm{In}_d)(x) = f(\frac{x+d}{2})$ and $I_d = [\frac{d-1}{2}, \frac{d+1}{2}]$ for each $d \in \{-1, 0, 1\}$. We call I_d a basic interval. We inductively define a predicate Read_X of arity (φ) by the following clauses

$$\forall_f^{\mathrm{nc}} \forall_d (f[I] \subseteq I_d \to X(\mathrm{Out}_d \circ f) \to \mathrm{Read}_X f), \qquad (\mathrm{Read}_X)_0^+$$

$$\forall_f^{\mathrm{nc}} (\mathrm{Read}_X(f \circ \mathrm{In}_{-1}) \to \mathrm{Read}_X(f \circ \mathrm{In}_0) \to \mathrm{Read}_X(f \circ \mathrm{In}_1) \to \mathrm{Read}_X f). \qquad (\mathrm{Read}_X)_1^+$$

The least-fixed-point axiom $(\mathrm{Read}_X)^-$ is defined to be

$$\begin{aligned} \forall_f^{\mathrm{nc}} (\mathrm{Read}_X f \to \forall_f^{\mathrm{nc}} \forall_d (f[I] \subseteq I_d \to X(\mathrm{Out}_d \circ f) \to Pf) \to \\ \forall_f^{\mathrm{nc}} (\mathrm{Read}_X(f \circ \mathrm{In}_{-1}) \to P(f \circ \mathrm{In}_{-1}) \to \\ \mathrm{Read}_X(f \circ \mathrm{In}_0) \to P(f \circ \mathrm{In}_0) \to \\ \mathrm{Read}_X(f \circ \mathrm{In}_1) \to P(f \circ \mathrm{In}_1) \to Pf) \to \\ Pf). \end{aligned} \qquad (\mathrm{Read}_X)^-$$

Furthermore, we give a nested inductive definition of a predicate Write of abstract uniformly continuous functions by the following clauses

$$\mathrm{Write}(\mathrm{Id}), \qquad \forall_f^{\mathrm{nc}} (\mathrm{Read}_{\mathrm{Write}} f \to \mathrm{Write} f),$$

where Id is the identity function. Witnesses for $\mathrm{Read}_X f$ and $\mathrm{Write} f$ are total ideals of $\mathbf{R}_\alpha$ and $\mathbf{W}$, respectively. We define ${}^{\mathrm{co}}\mathrm{Write}$, a companion predicate of Write, by the following clause

$$\forall_f^{\mathrm{nc}} ({}^{\mathrm{co}}\mathrm{Write} f \to f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}} f). \qquad ({}^{\mathrm{co}}\mathrm{Write})^-$$

The greatest-fixed-point axiom $({}^{\mathrm{co}}\mathrm{Write})^+$ of ${}^{\mathrm{co}}\mathrm{Write}$ is

$$\forall_f^{\mathrm{nc}} (Q\, f \to \forall_f^{\mathrm{nc}} (Q\, f \to f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write} \vee Q} f) \to {}^{\mathrm{co}}\mathrm{Write}\, f). \qquad ({}^{\mathrm{co}}\mathrm{Write})^+$$

A witness for ${}^{\mathrm{co}}\mathrm{Write} f$ is a $\mathbf{W}$-cototal $\mathbf{R}_{\mathbf{W}}$-total ideal. Intuitively, ${}^{\mathrm{co}}\mathrm{Write} f$ says that f is productive as a function on signed digit representations. If we can use axiom $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_0^+$, we know that the image of f is contained in an interval of radius $\frac{1}{2}$ centered at the digit d, so that the first output digit must be d independently of the input. By using the function $\mathrm{Out}_d \circ f$, we remove the leading digit and shift the input sequence one digit to the left. We continue to prove that f is productive on the rest of the input sequence. If the image of f is not contained in a basic interval, we can split the interval in three subintervals and check that f is productive on all of them by using axiom $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_1^+$. This corresponds to reading another input digit. Since Read_X is inductively defined, we can only use $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_1^+$ finitely many times before we are forced to use $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})_0^+$ and another output digit is determined.

In our Minlog formalization, φ is given as a type variable, and In_d, Out_d and Id are defined as constants without computational meaning with value type φ. This is not a problem, since all such constants will disappear in the program extraction process due to careful use of non-computational connectives.

3.3 Informal Proofs

We present informal proofs from which programs on uniformly continuous functions are extracted. Formalized proofs can be found in the Minlog distribution in the file `examples/analysis/readwrite.scm`.

For the first case study, Axiom 1 in Appendix A is used.

Theorem 1 (Type-1 u.c.f. into type-0 u.c.f.).

$$\forall_f^{\mathrm{nc}}(\mathrm{C}f \to {}^{\mathrm{co}}\mathrm{Write}f).$$

Proof. Let f be given and assume $\mathrm{C}f$. We prove ${}^{\mathrm{co}}\mathrm{Write}f$ by the greatest fixed point axiom ${}^{\mathrm{co}}\mathrm{Write}^+$ with C for the competitor. It suffices to prove $\forall_f^{\mathrm{nc}}(\mathrm{C}f \to f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}}f)$. Again let f be given and assume $\mathrm{C}f$, i.e. in particular $\mathrm{B}_{l,2}f$ for some l. By Lemma 2, the right disjunct of the goal holds. □

The above proof considerably depends on the following lemma, which in turn depends on the next ones.

Lemma 2. $\forall_l \forall_f^{\mathrm{nc}}(\mathrm{B}_{l,2}f \to \mathrm{C}f \to \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}}f)$.

Proof. By induction on l. *Base:* $l = 0$. Let f be given, and assume $\mathrm{B}_{0,2}f$ and $\mathrm{C}f$. Applying Lemma 3, there is a d such that $f[I] \subseteq I_d$. By Lemma 4, $\mathrm{C}f$ implies $\mathrm{C}(\mathrm{Out}_d \circ f)$, hence $({}^{\mathrm{co}}\mathrm{Write} \vee \mathrm{C})(\mathrm{Out}_d \circ f)$. Now use the introduction axiom $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}})_0^+$. *Step:* $l \mapsto l+1$. Suppose the following induction hypothesis

$$\forall_f^{\mathrm{nc}}(\mathrm{B}_{l,2}f \to \mathrm{C}f \to \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}}f), \tag{5}$$

and prove $\forall_f^{\mathrm{nc}}(\mathrm{B}_{l+1,2}f \to \mathrm{C}f \to \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}}f)$. Assume $\mathrm{B}_{l+1,2}f$ and $\mathrm{C}f$ for a given f. Our goal is $\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}}f$. By Lemma 4, we have $\mathrm{B}_{l,2}(f \circ \mathrm{In}_d)$ and $\mathrm{C}(f \circ \mathrm{In}_d)$ for each d. The induction hypothesis (5) yields $\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}}(f \circ \mathrm{In}_d)$ for each d, hence we can apply the introduction axiom $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}\vee\mathrm{C}})_1^+$ to finish the proof. □

Lemma 3. $\forall_f^{\mathrm{nc}}(\mathrm{B}_{0,2}f \to \exists_d(f[I] \subseteq I_d))$.

Proof. Assume f and $\mathrm{B}_{0,2}f$. From the definition of $\mathrm{B}_{l,k}$, $f[I_{0,0}] \subseteq I_{q,2}$ for some q holds. Because q is a rational number, either $q \le -\frac{1}{4}$, $-\frac{1}{4} \le q \le \frac{1}{4}$ or $\frac{1}{4} \le q$. Recall that our uniformly continuous function is bounded in $[-1,1]$. It is possible to determine either of $I_{q,2} \subseteq I_{-1}$, $I_{q,2} \subseteq I_0$ or $I_{q,2} \subseteq I_1$, hence $\exists_d(f[I] \subseteq I_d)$. □

Lemma 4. *(i)* $\forall_{f,k,l}^{\mathrm{nc}} \forall_d(f[I] \subseteq I_d \to \mathrm{B}_{l,k+1}f \to \mathrm{B}_{l,k}(\mathrm{Out}_d \circ f))$.
(ii) $\forall_f^{\mathrm{nc}} \forall_d(f[I] \subseteq I_d \to \mathrm{C}f \to \mathrm{C}(\mathrm{Out}_d \circ f))$.
(iii) $\forall_{f,k,l}^{\mathrm{nc}} \forall_d(\mathrm{B}_{l+1,k}f \to \mathrm{B}_{l,k}(f \circ \mathrm{In}_d))$.
(iv) $\forall_f^{\mathrm{nc}} \forall_d(\mathrm{C}f \to \mathrm{C}(f \circ \mathrm{In}_d))$. □

We now turn to calculating the definite integral of uniformly continuous functions to an arbitrary precision. In order to stay in the interval $[-1,1]$, we compute the definite integral from -1 to 1 divided by two. We abbreviate $\frac{1}{2}\int_{-1}^{1} f$ by $\int^{\mathrm{H}} f$ (H for "half"). The properties we need of the integral and the real numbers are axiomatized in Axiom 2 and Axiom 3 in Appendix A.

Theorem 5 (Definite integral from -1 to 1).

$$\forall_f^{\mathrm{nc}}({}^{\mathrm{co}}\mathrm{Write}f \to \forall_n \exists_p(\textstyle\int^{\mathrm{H}} f \in I_{p,n})).$$

Proof. Let f be given and assume ${}^{\mathrm{co}}\mathrm{Write}f$. We finish the proof by induction on n. *Case* $n = 0$. Choose p to be 0; then $\int^{\mathrm{H}} f \in I_{0,0}$ by the axiom. *Case* $n \mapsto n+1$. We prove $\exists_p(\int^{\mathrm{H}} f \in I_{p,n+1})$. By $({}^{\mathrm{co}}\mathrm{Write}f)^-$, we can do case distinction on $f = \mathrm{Id} \vee \mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}}f$. *Left case.* Suppose $f = \mathrm{Id}$. Let p be 0, then our goal is $\int^{\mathrm{H}} \mathrm{Id} \in I_{0,n+1}$, which is clear by the axioms. *Right case.* Suppose $\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}}f$ and use $(\mathrm{Read}_{{}^{\mathrm{co}}\mathrm{Write}})^-$. *Side base case.* Let f and d be given and assume $f[I] \subseteq I_d$ and ${}^{\mathrm{co}}\mathrm{Write}(\mathrm{Out}_d \circ f)$. We prove $\exists_p(\int^{\mathrm{H}} f \in I_{p,n+1})$. By i.h. there is a p' such that $\int^{\mathrm{H}}(\mathrm{Out}_d \circ f) \in I_{p',n}$, which implies $\int^{\mathrm{H}} f \in I_{\frac{p'+d}{2},n+1}$ as desired. *Side step case.* Let f be given and assume side i.h. We prove $\exists_p(\int^{\mathrm{H}} f \in I_{p,n+1})$. By the side i.h., there are p_d such that $\int^{\mathrm{H}}(f \circ \mathrm{In}_d) \in I_{p_d,n+1}$ for each d, thus $\frac{1}{2}(\int^{\mathrm{H}}(f \circ \mathrm{In}_{-1}) + \int^{\mathrm{H}}(f \circ \mathrm{In}_1)) \in I_{\frac{p_{-1}+p_1}{2},n+1}$ holds. This implies $\int^{\mathrm{H}} f \in I_{\frac{p_{-1}+p_1}{2},n+1}$ as desired. □

3.4 Extraction

From a proof, Minlog extracts a term in an extension of Gödel's T. In the next stage, these, together with relevant algebras and program constants, can be translated into a Haskell program using the `term-to-haskell-program` function of Minlog. We present the Haskell programs obtained from our formalized proofs. For aesthetic reasons, we present slightly formatted versions of the programs as suggested by e.g. HLint [19].

The algebras involved get translated to the following Haskell data types:

```
data AlgB = CInitB (Rational -> Rational)

data AlgRead a = Put Sd a
               | Get (AlgRead a) (AlgRead a) (AlgRead a)
  deriving (Show, Read, Eq, Ord, Functor)

data AlgWrite = Stop | Cont (AlgRead AlgWrite)
  deriving (Show, Read, Eq, Ord)

data Sd = L | M | R
  deriving (Show, Read, Eq, Ord)
```

We see how `AlgB` is just an isomorphic copy of `Rational -> Rational`, and how `AlgRead` has a type parameter `a` which gets instantiated to `AlgWrite` in the constructor `Cont`.

The extracted program from Lemma 3 is

```
cLemmaThree :: AlgB -> Sd
cLemmaThree (CInitB g) =
  if (numerator ((g 0) + (1/4)) > 0) then
    if (numerator ((g 0) - (1/4)) > 0) then R else M
  else L
```

This program computes a signed digit d such that the image of f – an abstract function which does not appear in the extracted term – is contained in I_d. It takes a rational function `g` which realizes $\mathrm{B}_{l,2}\, f$ as input; hence the image is contained in an interval of length $\frac{1}{2}$, centered at `g 0`. The calculation of the output is reduced to a simple decision of rational inequalities.

The extracted program from Lemma 2 is

```
cLemmaTwo :: Integer -> AlgB -> (Integer -> (Integer, AlgB)) ->
             AlgRead (Either AlgWrite
                             (Integer -> (Integer, AlgB)))
cLemmaTwo n =
  natRec n
         (\ w h ->
             Put (cLemmaThree w)
                 (Right (cLemmaFour_ii (cLemmaThree w) h)))
         (\ n3 g w h ->
             Get (g (cLemmaFour_iii L w) (cLemmaFour_iv L h))
                 (g (cLemmaFour_iii M w) (cLemmaFour_iv M h))
                 (g (cLemmaFour_iii R w) (cLemmaFour_iv R h)))
```

The extracted term `cLemmaTwo` takes as input a natural number `n`, a rational function `w` and a function $\mathtt{h} : \mathbf{N} \to \mathbf{N} \times \mathtt{AlgB}$ (in our application, we only call `cLemmaTwo` with $\langle \mathtt{n}, \mathtt{w} \rangle = \mathtt{h}\, 2$). Using recursion over `n`, it computes an approximation of `h` by a complete tree of height `n` with $3^{\mathtt{n}}$ leaves – a $\mathbf{R}_{\mathbf{W}+(\mathbf{N}\to\mathbf{N}\times\mathtt{AlgB})}$-total ideal. At the leaves, a signed digit d – computed from `w` using `cLemmaThree` – and the remainder of the approximation of `h` – computed by `cLemmaFour_ii` below, using d – is stored. At internal branching nodes, we split the domain of `h` into three subdomains – left, middle and right – modify `w` and `h` accordingly (using `cLemmaFour_iii` and `cLemmaFour_iv` below), and recurse.

The above term involves terms extracted from Lemma 4. They work in the following ways.

```
cLemmaFour_i :: Sd -> AlgB -> AlgB
cLemmaFour_i sd (CInitB h) =
  CInitB (\ a -> (2 * h a) - (sDToInt sd % 1))

cLemmaFour_ii :: Sd -> (Integer -> (Integer, AlgB)) ->
                 Integer -> (Integer, AlgB)
cLemmaFour_ii sd g n = case g (n + 1) of
  (n1, w1) -> (n1 , cLemmaFour_i sd w1)
```

```
cLemmaFour_iii :: Sd -> AlgB -> AlgB
cLemmaFour_iii sd (CInitB h) =
  CInitB (\ a -> h ((a + (sDToInt sd % 1)) / 2))

cLemmaFour_iv :: Sd -> (Integer -> (Integer, AlgB)) ->
                 Integer -> (Integer, AlgB)
cLemmaFour_iv sd g n = case g n of
  (n1, w) -> if n1 == 0 then (0, cLemmaFour_iii sd w)
             else (n1 - 1, cLemmaFour_iii sd w)
```

The extracted term from Theorem 1 is

```
type1to0 :: (Integer -> (Integer, AlgB)) -> AlgWrite
type1to0 r = algWriteCoRec r
  (\ h -> (Just (case h 2 of (n, w) -> cLemmaTwo n w h)))
```

This program corecursively constructs a $\mathbf{W}$-cototal $\mathbf{R}_{\mathbf{W}}$-total ideal by stacking $\mathbf{R}_{\mathbf{W}}$-total ideals computed by `cLemmaTwo`. It uses the modulus of continuity `n` at precision 2 to calculate an approximation of `s` as an $\mathbf{R}_{\mathbf{W}}$-total ideal, as in Lemma 2. In fact, `n` is the number of input signed digits to be read to determine one output signed digit. The extracted term from Theorem 5 is

```
integration :: AlgWrite -> Integer -> Rational
integration h n = natRec n (const 0)
  (\ n1 t h1 ->
    (case algWriteDestr h1 of
      Nothing -> 0
      Just s  -> algReadRec s
                 (\ sd h2 -> (t h2 + (sDToInt sd % 1)) / 2)
                 (\ s1 a1 s2 a2 s3 a3 -> (a1 + a3) / 2)))
  h
```

This program reads the given type-0 function to accumulate the possible output digits to compute the definite integral. The second argument is a number `n` to specify the bound of the computation in such a way that the program processes the read-write machine from its root up to the `n`th $\mathbf{R}_{\mathbf{W}}$-total ideals. At a branch, the recursively computed integral on the middle interval, i.e. `a2`, is ignored because it suffices to see value on the left and the right subintervals, i.e. $[-1, 0]$ and $[0, 1]$. At a leaf, the output digit is counted to contribute to the output with its height in the tree.

3.5 Experiment

As a first example, we instantiate the theorems to the function $f(x) \coloneqq -x$. From a type-1 representation of f, we compute a type-0 representation by means of our extracted program. We define f by $\langle h, \alpha, \omega \rangle$ where $h\, a\, n \coloneqq -a$, $\alpha\, n \coloneqq 0$ and $\omega\, n \coloneqq n + 1$. The input of `type1to0` is $\lambda_n \langle \omega n, \lambda_a(ha(\alpha n)) \rangle$ which turns

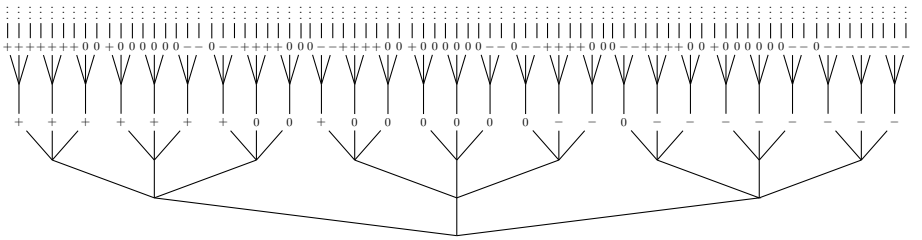


Fig. 1. Type-0 representation of $f(x) = -x$

into the Haskell expression `fIn = \ n -> (n+1, CInitB (\ x -> -x))`. The output `type1to0 fIn` is graphically presented in Figure 1, where Cont is omitted, Get is a branching node and Put d is denoted by $-$, 0 or $+$ respectively for $d = -1, 0$ or 1.

In our next example, we compute half of the definite integral for the function $f(x) := \sqrt{x+2} - 1$. This integrand is defined to be $\langle h, \alpha, \omega \rangle$, where $h\, a\, n :=$ $(\mathcal{R}_{\mathbf{N}}\, n\, 1\, \lambda_{\_,b}\, (\frac{b+\frac{a+2}{b}}{2})) - 1$, $\alpha\, n := n+1$, and $\omega\, n :=$ Pred n where Pred : $\mathbf{N} \to \mathbf{N}$ is the predecessor function. Converting this to the Haskell function it represents, we end up with `\ n -> (Main.pred n, CInitB (\ x -> h x (n+1)))`. We give two arguments to `integration`, a type-0 function and a natural number, the accuracy. The first input to `integration` is computed by `type1to0` from the above type-1 function. Specifying 8 as the second argument, the output is `1633 % 4096` whose decimal expansion is $0.398681640625\cdots$. Comparing our result with the manually calculated definite integral $\frac{1}{2}\int_{-1}^{1} f(x)dx = \sqrt{3} - \frac{4}{3}$, the error is $0.00003583\cdots$, which indeed is smaller than $2^{-8} = 0.00390625$.

4 Conclusion

We presented the formal theory TCF and its implementation Minlog which support nested inductive/coinductive definitions. Minlog extracts programs in an extension of Gödel's T from proofs involving nested definitions. Moreover, terms in the extension of Gödel's T can be translated into programs in programming languages such as Haskell. We gave an application to the theory of uniformly continuous functions as an illustration.

Related Work. Nested definitions are used by Ghani, Hancock and Pattinson [11,12] to define uniformly continuous functions. They are also studied by Bird and Meertens [6] from a purely programming perspective. Krebbers and Spitters [15] give effective certified programs for exact real number computation. Berger and Seisenberger [4] considers "pen and paper" program extraction for a system with induction and coinduction. Berger [3] studies program extraction and its application to exact real arithmetic. He manually extracts programs

from proofs dealing with uniformly continuous functions. Our case study is heavily based on his results. More case studies based on Berger's work, e.g. function application and composition, are available in the Minlog distribution. Also other researchers have studied the combination of induction and coinduction. Nakata and Uustalu [20] study the semantics of interactive programs by means of induction nested into coinduction, and give a formalization in Coq. Danielsson and Altenkirch [10] study so-called mixed induction and coinduction, using Agda.

References

1. Agda, `http://wiki.portal.chalmers.se/agda/`
2. Berger, U.: Program extraction from normalization proofs. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 91–106. Springer, Heidelberg (1993)
3. Berger, U.: From coinductive proofs to exact real arithmetic: theory and applications. Logical Methods in Computer Science 7(1), 1–24 (2011)
4. Berger, U., Seisenberger, M.: Proofs, programs, processes. Theory of Computing Systems 51, 313–329 (2012)
5. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 24–40. Springer, Heidelberg (2002)
6. Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 52–67. Springer, Heidelberg (1998)
7. Bishop, E.: Foundations of Constructive Analysis. McGraw-Hill, New York (1967)
8. Ciaffaglione, A., Di Gianantonio, P.: A co-inductive approach to real numbers. In: Coquand, T., Nordström, B., Dybjer, P., Smith, J. (eds.) TYPES 1999. LNCS, vol. 1956, pp. 114–130. Springer, Heidelberg (2000)
9. Coq, `http://coq.inria.fr/`
10. Danielsson, N.A., Altenkirch, T.: Mixing Induction and Coinduction. Draft (2009)
11. Ghani, N., Hancock, P., Pattinson, D.: Continuous functions on final coalgebras. In: Power, J. (ed.) CMCS 2006. Electr. Notes in Theoret. Computer Science (2006)
12. Hancock, P., Pattinson, D., Ghani, N.: Representations of stream processors using nested fixed points. Logical Methods in Computer Science 5(3) (2009)
13. Isabelle, `http://isabelle.in.tum.de/`
14. Jacobs, B., Rutten, J.: An introduction to (co)algebras and (co)induction. In: Sangiorgi, D., Rutten, J. (eds.) Advanced Topics in Bisimulation and Coinduction, vol. 52, pp. 38–99. Cambridge University Press (2011)
15. Krebbers, R., Spitters, B.: Type classes for efficient exact real arithmetic in Coq. Logical Methods in Computer Science 9(1) (2013)
16. Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. In: Heyting, A. (ed.) Constructivity in Mathematics, pp. 101–128. North-Holland, Amsterdam (1959)
17. Letouzey, P.: Extraction in coq: An overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008)
18. The Minlog System, `http://www.minlog-system.de`
19. Mitchell, N.: HLint, `http://community.haskell.org/~ndm/hlint/`
20. Nakata, K., Uustalu, T.: Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: An exercise in mixed induction-coinduction. In: Aceto, L., Sobocinski, P. (eds.) SOS. EPTCS, vol. 32, pp. 57–75 (2010)

21. Nuprl, http://www.nuprl.org/
22. Schwichtenberg, H.: Minlog. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. LNCS (LNAI), vol. 3600, pp. 151–157. Springer, Heidelberg (2006)
23. Schwichtenberg, H.: Constructive analysis with witnesses. Manuscript (April 2012), http://www.math.lmu.de/~schwicht/seminars/semws11/constr11.pdf
24. Schwichtenberg, H., Wainer, S.S.: Proofs and Computations. In: Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press (2012)
25. Scott, D.: Domains for denotational semantics. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 577–610. Springer, Heidelberg (1982)
26. Takeyama, M.: A new compiler MAlonzo, https://lists.chalmers.se/pipermail/agda/2008/000219.html
27. Wiedmer, E.: Exaktes Rechnen mit reellen Zahlen und anderen unendlichen Objekten. PhD thesis, ETH Zürich (1977)
28. Wiedmer, E.: Computing with infinite objects. Theoretical Comput. Sci. 10, 133–155 (1980)

A Axioms

Axiom 1 (Abstract Theory of Uniformly Continuous Functions)

$$\forall_{f,d,p,l,q,k}(f[I] \subseteq I_d \to \mathrm{Out}_d \circ f[I_{p,l}] \subseteq I_{q,k} \to f[I_{p,l}] \subseteq I_{\frac{q+d}{2},k+1}), \tag{OutElim}$$

$$\forall_{f,d,p,l,q,k}(f[I] \subseteq I_d \to f[I_{p,l}] \subseteq I_{\frac{q+d}{2},k+1} \to \mathrm{Out}_d \circ f[I_{p,l}] \subseteq I_{q,k}), \tag{OutIntro}$$

$$\forall_{f,d,p,l,q,k}(f \circ \mathrm{In}_d[I_{p,l}] \subseteq I_{q,k} \to f[I_{\frac{p+d}{2},l+1}] \subseteq I_{q,k}), \tag{InElim}$$

$$\forall_{f,d,p,l,q,k}(f[I_{\frac{p+d}{2},l+1}] \subseteq I_{q,k} \to f \circ \mathrm{In}_d[I_{p,l}] \subseteq I_{q,k}), \tag{InIntro}$$

$$\forall_{f,p}(f[I_{p,0}] \subseteq I), \tag{UcfBound}$$

$$\forall_{p,l}(\mathrm{Id}[I_{p,l}] \subseteq I_{p,l}), \tag{UcfId}$$

$$\forall_{f,p,l,q,k}(f[I_{p,l}] \subseteq I_{q,k} \to f[I_{p,l+1}] \subseteq I_{q,k}), \tag{UcfInputSucc}$$

$$\forall_{f,q}(q \le -\tfrac{1}{4} \to f[I] \subseteq I_{q,2} \to f[I] \subseteq I_{-1}), \tag{UcfLeft}$$

$$\forall_{f,q}(-\tfrac{1}{4} \le q \le \tfrac{1}{4} \to f[I] \subseteq I_{q,2} \to f[I] \subseteq I_0), \tag{UcfMiddle}$$

$$\forall_{f,q}(\tfrac{1}{4} \le q \to f[I] \subseteq I_{q,2} \to f[I] \subseteq I_1). \tag{UcfRight}$$

Axiom 2 (Abstract Theory of Real Numbers)

$$\forall_n(0 \in I_{0,n}), \tag{RealZero}$$

$$\forall_{x,p,n,d}(x \in I_{p,n} \to \tfrac{x+d}{2} \in I_{\frac{p+d}{2},n+1}), \tag{AvIntro}$$

$$\forall_{x,y,p,q,n}(x \in I_{p,n} \to y \in I_{q,n} \to \tfrac{x+y}{2} \in I_{\frac{p+q}{2},n}). \tag{RealAvrg}$$

Axiom 3 (Abstract Theory of Integration)

$$\forall_{f,d}(\textstyle\int^{\mathrm{H}} f = \frac{1}{2}(\int^{\mathrm{H}}(\mathrm{Out}_d \circ f) + d)), \tag{HIntOut}$$

$$\forall_f(\textstyle\int^{\mathrm{H}} f = \frac{1}{2}(\int^{\mathrm{H}}(f \circ \mathrm{In}_{-1}) + \int^{\mathrm{H}}(f \circ \mathrm{In}_1))), \tag{HIntIn}$$

$$\textstyle\int^{\mathrm{H}} \mathrm{Id} = 0, \tag{HIntId}$$

$$\forall_f(\textstyle\int^{\mathrm{H}} f \in I_{0,0}). \tag{HIntBound}$$

Subformula Linking as an Interaction Method

Kaustuv Chaudhuri

INRIA, France
http://chaudhuri.info

Abstract. Current techniques for building formal proofs interactively involve one or several proof languages for instructing an interpreter of the languages to build or check the proof being described. These linguistic approaches have a drawback: the languages are not generally portable, even though the nature of logical reasoning is universal. We propose a somewhat speculative alternative method that lets the user directly manipulate the text of the theorem, using non-linguistic metaphors. It uses a proof formalism based on linking subformulas, which is a variant of deep inference (inference rules are allowed to apply in any formula context) where the relevant formulas in a rule are allowed to be arbitrarily distant. We substantiate the design with a prototype implementation of a linking-based interactive prover for first-order classical linear logic.

1 Introduction

Formal proofs are more fundamentally computational than mathematical. Say we want to prove $(a \supset b) \supset (b \supset c) \supset (a \supset c)$. Current proof languages are generally always languages of *interaction*, so a proof is an arrangement of instructions to a stateful interpreter that allows it to build (or check) an underlying proof object in a trusted natural deduction or a sequent calculus formulation of the logic. A formal proof of this theorem is, therefore, something like this;

> Suppose (1) $a \supset b$, (2) $b \supset c$, (3) a; to show c, we backchain (2) to change the goal to b, then we backchain (1) to change it to a, which we already have by (3).

(Most proof languages can express the above more succinctly.) This proof naturally resembles a computational trace where the steps of the computation are the (primitive or derived) inference rules of the logic, possibly with the use of additional lemmas. However, this formal proof obscures the simple mathematical intuition behind the proof, function composition; indeed, the composition is fully unfolded and sequentialized. In order to write the more natural mathematical proof, the user would first have to find a lemma in the standard library (or write one) that implements this intuition.

Because formal proof languages are more closely tied to the underlying calculus, and often to formal theories, they are generally not portable as such across different systems. In the rare instances where two different formal proof systems (take, *e.g.*, Coq and Isabelle) do exchange proof objects, they tend to be low-level "logical bytecode" (λ-terms built internally by the proof tactics or rewriting

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 386–401, 2013.

traces) that bear no resemblance to the proof text a user actually writes and are not expected to be read by humans (some examples: [17,5]). Only the most committed (and masochistic) user learns to be proficient in many systems, so the communities of users of formal systems have become rather balkanized. It is worth asking if there is a sense in which one can build an interactive prover free of the shackles of language.

To answer this question, let us begin by noting that a proof fundamentally consists of two varieties of reasoning: reasoning about the subformula *structure*, and reasoning about *links* between different subformulas. Examples of the first variety are: to prove the goal $A \wedge B$ create two independent subgoals for A and B, or to use an assumption $\forall x.\, A$ add an instantiation $[t/x]A$ as an additional assumption. Examples of the second variety are: to prove the goal A search for an assumption A (use of hypothesis), or to prove the goal C first prove A and then show that C follows from A (cut or substitution). The language for establishing the links is surprisingly similar across proof languages; they generally always use hypothesis labels and terms such as `apply` or `assert`. Structural reasoning differs widely, however; at one end of a spectrum are languages based on tactics that drive a stateful prover through a proof tree, while at the other end are languages that reify the proof tree textually in the proof document but do not prescribe an order of evaluation. Crucially, though, the entire point of the structural reasoning is to enable the links, which are the only ways to finish and compose proofs and are therefore the essence of a proof, so the surprising design focus on structural reasoning seems misplaced.

How can one make linking the primary aspect of formal proof construction? Unsurprisingly, this question has been asked and answered a number of times, but never satisfactorily. This paper combines two existing answers—*proof by pointing* [4] and the *calculus of structures* [7,15]—in a way that improves on both and suggests a research direction.

Proof by pointing is the first attempt to systematically construct a non-linguistic interaction method on top of an existing proof interaction language such as the tactics language of Coq [3]. The user proves a theorem by a sequence of mouse clicks on subformulas; each click brings the indicated formula to the foreground of the current goal, and possibly creates additional background subgoals. Depending on the occurrence of the subformula, each click is interpreted as a sequence of tactics that mimics a sequence of sequent rules. Once a subformula is brought into the foreground on both sides of the sequent arrow, the corresponding link is established and the goal is closed. The problem with the pointing approach appears after this closure: the remaining subgoals that were produced to establish the link are left in an "exposed" state and may be difficult to reason about subsequently. To illustrate, in the earlier example of $(a \supset b) \supset (b \supset c) \supset (a \supset c)$ the user can click on (*i.e.*, point to) the two occurrences of b. The residual goals after the main goal is closed are: $a \supset b \vdash a$ and $a \supset b, b \supset c, c \vdash a \supset c$; the former goal is unprovable! This does not correspond to the intuition of linking the two bs, *i.e.*, of composing the assumptions $a \supset b$ and $b \supset c$ to get a new assumption $a \supset c$, which would match the succedent of the original theorem.

The problem with the example above is that interpreting clicks as applications of sequent rules destroys the surrounding formula context by bringing the indicated subformula to the top of the sequent. The interaction method is inherently *deep* (working on arbitrary subformulas), but the underlying shallow calculus prevents deep reasoning. The user is forced to think about the order of clicks to ensure that the sequents have the right shape. For instance, in the above example the user would have to click on the rightmost a first, which would ensure that when the bs are finally clicked on that the first residual subgoal is not $a \supset b \vdash a$ but $a, a \supset b \vdash a$.

To escape the bureaucracy of sequential syntax, we require a calculus of *deep inference* that can preserve formula contexts and thereby perform logically unrelated reasoning steps in a free order. In this paper we use the *calculus of structures* where there is no difference between a formula and a sequent, and proof steps are allowed to apply in any formula context. Unfortunately, the calculus of structures by itself cannot support our desired linking procedure above either, because every rule in the calculus is still limited to immediate operands of the principal connectives. (The two bs in the example were only ancestrally linked.) We need to generalize the calculus of structures so that the interacting subformulas may be themselves separated by other subformulas.

As the main technical contribution, this paper gives a calculus for *linked structures* for first-order classical linear logic (Sec. 3). We pick linear logic because: (1) it seamlessly encodes many intuitionistic and classical logics and has clean proof-theoretic foundations; (2) it directly represents resource-nondeterministic choices (splitting and sharing) that are essential when reasoning about computations; and (3) it is an extension to proof by pointing that was left to future work in [4,11]. We substantiate the claims in the paper by a prototype implementation of a linking-based proof-assistant called *Profound* (Sec. 4). Our implementation is not currently integrated with any mainstream theorem proving system, so it should be seen as conceptual and somewhat speculative.

2 The Calculus of Structures for Linear Logic

Let us begin with an overview of the calculus of structures for classical first-order linear logic. Formulas (written $A, B, \ldots$) have the following grammar,

$$\begin{aligned} A, B, \ldots ::= {} & a \mid A \otimes B \mid \mathsf{1} \mid A \oplus B \mid \mathsf{0} \mid \mathsf{!}A \mid \exists x.\, A \\ & \mid \overline{a} \mid A \parr B \mid \bot \mid A \mathbin{\&} B \mid \top \mid \mathsf{?}A \mid \forall x.\, A \end{aligned}$$

Atomic formulas (or *atoms*) are written using $a, b, \ldots$, and the negation of a is written as $\overline{a}$. We use the term *literal* to refer to either an atom or a negated atom. Each atomic formula is of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol and $t_1, \ldots, t_n$ are first-order terms (written $s, t, \ldots$) formed from term variables (written $x, y, \ldots$) and applications of function symbols (written $f, g, \ldots$) to terms. As is standard, we assume an ambient *signature* for the logic that assigns arities to predicate and function symbols. Formulas are in negation-normal form with each vertical column in the above grammar depicting one De Morgan dual pair; we write $\overline{A}$ for the dual of A.

$$\frac{}{\vdash \overline{a}, a}\,\mathsf{init} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\,\mathsf{weak} \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\,\mathsf{contr}$$

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\otimes \qquad \frac{}{\vdash 1}\,1 \qquad \frac{\vdash \Gamma, A_i}{\vdash \Gamma, A_1 \oplus A_2}\oplus \qquad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\,! \qquad \frac{\vdash \Gamma, [t/x]A}{\vdash \Gamma, \exists x.\, A}\,\exists$$

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\,\& \qquad \frac{}{\vdash \Gamma, \top}\,\top \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\,\bot \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\,? \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, \forall x.\, A}\,\forall$$

Fig. 1. Rules of *LLK*. In the $\oplus$ rule, $i \in \{1, 2\}$. In the ! rule, $?\Gamma$ stands for a multiset of formulas each of which is prefixed by ?. In the $\forall$ rule, x is not free in the conclusion.

$$\frac{1}{a \parr \overline{a}}\,\mathsf{id} \qquad \frac{B \parr A}{A \parr B}\,\mathsf{com} \qquad \frac{(A \parr B) \parr C}{A \parr (B \parr C)}\,\mathsf{asc} \qquad \frac{A}{A \parr \bot}\,\mathsf{bot} \qquad \frac{\bot}{?A}\,\mathsf{wk} \qquad \frac{?A \parr ?A}{?A}\,\mathsf{con}$$

$$\frac{(A \parr C) \otimes B}{(A \otimes B) \parr C}\,\mathsf{lsp} \qquad \frac{B \otimes (A \parr C)}{A \parr (B \otimes C)}\,\mathsf{rsp} \qquad \frac{A_i}{A_1 \oplus A_2}\,\mathsf{chc} \qquad \frac{!(?A \parr B)}{?A \parr !B}\,\mathsf{bng} \qquad \frac{[t/x]A}{\exists x.\, A}\,\mathsf{wit}$$

$$\frac{(A \parr B) \,\&\, (A \parr C)}{A \parr (B \,\&\, C)}\,\mathsf{add} \qquad \frac{\top}{A \parr \top}\,\mathsf{top} \qquad \frac{A}{?A}\,\mathsf{drl} \qquad \frac{\forall x.\, (A \parr B)}{(\forall x.\, A) \parr B}\,\mathsf{all}$$

$$\frac{1}{1 \otimes 1} \qquad \frac{1}{1 \,\&\, 1} \qquad \frac{1}{\top} \qquad \frac{1}{!1} \qquad \frac{1}{\forall x.\, 1}$$

Fig. 2. Rules of *CLS*. In the chc rule, $i \in \{1, 2\}$. In the all rule, x is not free in B.

For our definition of *truth* in this logic, we use a standard one-sided cut-free sequent calculus—called system *LLK*—consisting of sequents of the form $\vdash \Gamma$, where Γ is a multiset of formulas. Figure 1 contains the rules of *LLK*. The meta-theory of this calculus can be found in any standard reference [10,19].

The calculus of structures is a formalism where there is no difference between formula and sequent. It is a rewrite system where some entailment $\vdash \overline{A}, B$ are turned into rewrite rules that replace, reading from conclusion to premises, a subformula B in a formula with A. The system is best described in a *contextual form*, where a *formula context* (written $\chi, \xi, \ldots$) is formed by replacing a single subformula of a formula by the *hole* ($\Box$). We write $\chi\{A\}$ for the formula formed by replacing the hole in χ by A (possibly capturing the free variables of A).

Figure 2 contains the rules of the classical linear system *CLS*, which is a minor variant of the system *LS* from [7]. Each rule is to be understood as closed over a formula context; to illustrate, the rule id represents this general schema.

$$\frac{\chi\{1\}}{\chi\{a \parr \overline{a}\}}$$

The first line of rules in Fig. 2 are the structural rules of the system. They are, in order, the atomic identity rule; rules for commutativity, associativity, and unit for $\parr$ (due to which $\langle \parr, \bot \rangle$ behaves as a commutative monoid structure); and weakening and contraction for ?-formulas. The second line contains logical rules

corresponding to the positive connectives: splits (lsp and rsp) for $\otimes$, choices (chc) for $\oplus$, promotion (bng), and instantiation (wit). In the wit rule, the witness term t is only allowed to mention the bound variables in scope at the hole; in other words, the signature over which conclusion and premise are well-formed does not change on the application of any rule. The third line of rules contain the logical rules for the negative connectives: distributivity (add) for &, absorption (top) for $\top$, dereliction (drl) for ?, and scope-enlargement of universal quantification (all). The final line of rules define the mechanisms of combining different successful "branches" of a *CLS* proof.

A *derivation* ϕ of B from A in any unary proof-system $\mathcal{S}$, written $\phi : A \xrightarrow{\mathcal{S}} B$, is a sequence of rules of $\mathcal{S}$ with the bottom-most rule having conclusion B and the top-most rule having premise A. We write just $A \xrightarrow{\mathcal{S}} B$ to assert that there is a ϕ such that $\phi : A \xrightarrow{\mathcal{S}} B$. A *proof* ϕ of A in *CLS* is a derivation $\phi : 1 \xrightarrow{CLS} A$. Under this notion of proof, *CLS* is sound and complete with respect to the usual linear truth given in terms of *LLK*.

Theorem 1 (Soundness of *CLS* wrt. *LLK*). *If* $A \xrightarrow{CLS} B$, *then* $\vdash \overline{A}, B$ *in LLK.*

Proof. Every inference rule in *CLS* with premise A and conclusion B corresponds to a provable sequent $\vdash \overline{A}, B$ of *LLK*. The result follows by a sequence of applications of cut in *LLK*. □

Theorem 2 (Completeness of *CLS* wrt. *LLK*). *If* $\vdash A_1, \ldots, A_n$ *in LLK, then* $1 \xrightarrow{CLS} A_1 \parr \cdots \parr A_n$.

Proof. Under the interpretation of an *LLK* sequent $\vdash A_1, \ldots, A_n$ as the formula $A_1 \parr \cdots \parr A_n$, every rule of *LLK* is derivable in *CLS*. Generally speaking, one *LLK* rule corresponds to a sequence of *CLS* rules; in particular, the $\otimes$ rule of *LLK* corresponds to a sequence of applications of lsp and rsp, while the & rule corresponds to a sequence of applications of add. □

Although the system *CLS* is cut-free, the following cut rule is admissible:

$$\frac{\chi\{A \otimes \overline{A}\}}{\chi\{\bot\}}\,\mathsf{cut}$$

As usual, this rule is just the dual of the generalized identity (gid) rule:

$$\frac{\chi\{1\}}{\chi\{A \parr \overline{A}\}}\,\mathsf{gid}$$

Indeed, every rule of *CLS* has a dual form where the premise and conclusion are exchanged and dualized. The dual system of *CLS*, which we write as $\overline{CLS}$, is a system of *refutation*, where a refutation of A is a proof $A \xrightarrow{\overline{CLS}} \bot$, and the system that contains both *CLS* and $\overline{CLS}$ is a system where both cut and identity can be reduced to their atomic forms [7,15]. Note that the rules lsp and rsp are self-duals, and therefore in the intersection of *CLS* and its dual system.

3 Formula Linking

The system *CLS* forms the foundation of the underlying calculus of the formula linking approach. As mentioned in the introduction, we would like to support a mode of interaction where the user can indicate that any two subformulas are to be linked. The rules of *CLS* are not well suited for such interactions because each inference rules operates on two immediately $\parr$-joined formulas.

The essence of linking is to generalize the interaction from $\parr$-joined formulas to formulas that are (eventually) ancestrally joined by $\parr$. Then, the user can simply mark the formulas they wish to link and the system will apply the right sequence of *CLS* rules to bring the linked formulas into the same context. If the linked formulas are dual literals, then the system can simply replace the $\parr$-formula with 1. The user can therefore repeatedly link dual literals when constructing a proof. (The user is of course free to link other non-atomic subformulas as well.) Importantly, the user is allowed to make incorrect linkages that render the formula unprovable. We require only that any provable formula be also provable by a linkage simplification procedure.

First, let us make the notion of linkage formal. We enlarge the syntax of formulas with a new form, $\mathfrak{u}::A$, called a *linkage*, which indicates that the formula A is *marked* by a *link* $\mathfrak{u}$. (Links are drawn from some infinite set that is disjoint from the set of variables.) We write $\mathfrak{u} \in C$ to assert that C contains a subformula of the form $\mathfrak{u}::A$. We require that every linkage in a formula occurs *exactly twice*, and that the two subformulas marked by a link be ancestrally joined by $\parr$ in the subformula ordering. In other words, for each link $\mathfrak{u} \in C$, there must exist formula contexts ξ, χ_1, χ_2 and subformulas A and B of C such that $C = \xi\{\chi_1\{\mathfrak{u}::A\} \parr \chi_2\{\mathfrak{u}::B\}\}$. These restrictions will be preserved by the linkage simplification rules.

The user initiates a linkage by marking two distinct ancestrally $\parr$-joined subformulas of the goal formula by a fresh link. We write this as an inference rule that produces a new kind of connective $*$ (called *interaction*), as follows:

$$\frac{\xi\{\chi_1\{\mathfrak{u}::A\} * \chi_2\{\mathfrak{u}::B\}\}}{\xi\{\chi_1\{A\} \parr \chi_2\{B\}\}}\,\mathsf{lnk}$$

We restrict this rule to only be applicable when the conclusion formula is free of all linkages, although this restriction can often be relaxed. Note that unlike the rules of *CLS* (Fig. 2), the rule lnk is *doubly deep*: not only is the principal formula allowed to occur in any formula context, but also the result of applying the rule has an effect on subformulas that are not necessarily the immediate descendants of the principal formula.

Semantically, $*$ has the same truth value as $\parr$, but the rules applicable to $*$ have more restrictions than the analogous rules for $\parr$. If the immediate operands of the interaction are the linkages—*i.e.*, if the context χ_1 and χ_2 in the rule lnk above are just holes—then one of the following two rules will be applied.

$$\frac{\chi\{1\}}{\chi\{\mathfrak{u}::a * \mathfrak{u}::\overline{a}\}}\,\mathsf{lnid} \qquad \frac{\chi\{A \parr B\}}{\chi\{\mathfrak{u}::A * \mathfrak{u}::B\}}\,\mathsf{unlnk}$$

where the unlnk rule is only applicable when A and B are not dual literals.

The remaining rules for interaction connectives are used to permute it into smaller contexts. Defining these rules requires a bit of caution, as best illustrated by an example.

Example 3. Let us temporarily ignore linkages and just consider the interaction connective $*$. Consider the provable formula $(a \otimes b) \parr (\overline{a} \mathbin{\&} \overline{b}) \parr (\overline{a} \oplus \overline{b})$. Suppose the common $\parr$-ancestor a and the leftmost $\overline{a}$ is turned into $*$ with lnk.

$$\frac{\big((a \otimes b) * (\overline{a} \mathbin{\&} \overline{b})\big) \parr (\overline{a} \oplus \overline{b})}{(a \otimes b) \parr (\overline{a} \mathbin{\&} \overline{b}) \parr (\overline{a} \oplus \overline{b})}\,\mathsf{lnk}$$

Let us attempt to use the same rules as $\parr$ in *CLS* for the $*$-formulas. If we immediately apply a lsp to the interaction formula in the premise above, we would obtain an unprovable formula:

$$\frac{\big((a * (\overline{a} \mathbin{\&} \overline{b})) \otimes b\big) \parr (\overline{a} \oplus \overline{b})}{\big((a \otimes b) * (\overline{a} \mathbin{\&} \overline{b})\big) \parr (\overline{a} \oplus \overline{b})}\,\mathsf{lsp}$$

The issue here is that lsp forces both $\overline{a}$ and $\overline{b}$ (that are &-joined) to be "sent" to exactly one of the operands of the $\otimes$, but the theorem is only provable if exactly one of the $\otimes$-operands gets exactly one of the &-operands. This can be achieved by a different order of the inference rules, performing add before lsp as follows.

$$\dfrac{\dfrac{\big(((a * \overline{a}) \otimes b) \mathbin{\&} (a \otimes (b * \overline{b}))\big) \parr (\overline{a} \oplus \overline{b})}{\big(((a \otimes b) * \overline{a}) \mathbin{\&} ((a \otimes b) * \overline{b})\big) \parr (\overline{a} \oplus \overline{b})}\,\mathsf{lsp}\text{ and }\mathsf{rsp}}{\big((a \otimes b) * (\overline{a} \mathbin{\&} \overline{b})\big) \parr (\overline{a} \oplus \overline{b})}\,\mathsf{add}$$

The formula in the premise is now provable in *CLS* under the interpretation of $*$ as $\parr$. As a general principle, if there is more than one way to interpret a linkage, we should pick an interpretation that preserves provability.

3.1 Polarities as Organizational Hints

To build our provability-preserving interaction rules, we use the well known concept of *polarity*. Some rules of *CLS* such as the split rules lsp and rsp or the choice rule chc add information to the proof; they correspond to proper implications and are not invertible as rules. Other rules such as add or all are equivalences (the conclusion implies the premise), and therefore the rules are invertible. Following general terminology, we say that the connectives with non-invertible interactions have *positive* polarity, while those with invertible interactions have *negative* polarity. More precisely, the formulas are separated into *positive formulas* (written $P, Q, \dots$) and *negative formulas* (written $N, M, \dots$) according to the following grammar.

$$\begin{array}{lll} A, B, \dots & ::= & P \mid N \\ P, Q, \dots & ::= & a \mid A \otimes B \mid 1 \mid A \oplus B \mid 0 \mid !A \mid \exists x.\, A \\ N, M, \dots & ::= & \overline{a} \mid A \parr B \mid \bot \mid A \mathbin{\&} B \mid \top \mid ?A \mid \forall x.\, A \end{array}$$

$$\frac{(\chi\{\mathrm{u}::A\} * \rho\{\mathrm{u}::C\}) \otimes B}{(\chi\{\mathrm{u}::A\} \otimes B) * \rho\{\mathrm{u}::C\}}\,\mathsf{Inspl} \qquad \frac{B \otimes (\rho\{\mathrm{u}::A\} * \chi\{\mathrm{u}::C\})}{\rho\{\mathrm{u}::A\} * (B \otimes \chi\{\mathrm{u}::C\})}\,\mathsf{Inspr}$$

$$\frac{!(?\chi\{\mathrm{u}::A\} * \xi\{\mathrm{u}::B\})}{?\chi\{\mathrm{u}::A\} * !\xi\{\mathrm{u}::B\}}\,\mathsf{Inbng}$$

$$\frac{(\chi\{\mathrm{u}::A\} * \xi\{\mathrm{u}::C\}) \,\&\, (B \parr \xi\{C\})}{(\chi\{\mathrm{u}::A\} \,\&\, B) * \xi\{\mathrm{u}::C\}}\,\mathsf{Inaddl} \qquad \frac{(A \parr \xi\{C\}) \,\&\, (\chi\{\mathrm{u}::B\} * \xi\{\mathrm{u}::C\})}{(A \,\&\, \chi\{\mathrm{u}::B\}) * \xi\{\mathrm{u}::C\}}\,\mathsf{Inaddr}$$

$$\frac{\forall x.\, (\chi\{\mathrm{u}::A\} * \xi\{\mathrm{u}::C\})}{(\forall x.\, \chi\{\mathrm{u}::A\}) * \xi\{\mathrm{u}::C\}}\,\mathsf{Inall} \qquad \frac{B * A}{A * B}\,\mathsf{Incom}$$

Fig. 3. Rules of *CLn*

Note that the dual of a positive formula is a negative formula and *vice versa*.

The key feature of polarity is that the rules for positive connectives commute with those for other positive connectives, and likewise for negative connectives with other negative connectives, but the rules for positive and negative connectives do not necessarily commute with each other. It is common to use this observation to design a *focused* version of a sequent calculus with alternate positive and negative *phases* of rules [1,8,12]. Focusing drastically reduces the nondeterminism during proof search (see the experimental results in, *e.g.*, [8,13]), but it also clarifies the structure of sequent proofs by identifying, precisely, the informative portions of full sequent proofs. The calculus of structures can also be focused [7], although the phase structure is not as evident there because of the incremental and interleaved nature of deep inference. We can, of course, adopt the full focusing discipline in this paper, but the additional structure on proofs is sometimes counter-intuitive from a user's perspective. Instead, we will use features from focusing *solely* to handle the interaction connective $*$; in other words, $A \parr B$ is the ordinary unfocused formula, while $A * B$ denotes a *focused interaction* where, if either A or B is positive then it behaves as the analogue of a focused formula (called the *head* in [7]). Note that, unlike the system in [7], there can be zero, one, or two positive formulas in a focused interaction.

Figure 3 defines the system *CLn* of these focused rules for the interaction connective. In these rules, ρ represents a positive formula context, *i.e.*, for any formula A, the substitution $\rho\{A\}$ is either A itself or a positive formula. *CLn* can obviously be seen as a subsystem of *CLS* with the interpretation of $*$ as $\parr$ and ignoring the linkage information. In particular, it corresponds to that fragment of *CLS* that defines the interaction of two $\parr$-joined formulas. The number of occurrences of interaction formulas and marked subformulas is the same in both premise and conclusion in every rule of *CLn*. The add rule of *CLS* splits into two forms in *CLn*, depending on which of the operands of the $\&$ contains the linkage.

The system *CLn* defines all the rules for simplifying interactions. We embed it into a larger system, named *CLnS*, that contains:

- the rules lnk, lnid, and unlnk (from above);
- the rules of *CLn* (Fig. 3); and
- the *CLS* rules com, asc, bot, wk, con, chc, wit, top, drl and all the rules in the final line in Fig. 2.

CLnS is relatively complete with respect to *CLS*. This does not mean, of course, that any application of lnk will be valid; consider, for example, linking the a and the $\overline{a}$ in $!a \parr \overline{a}$. Instead, we have the following correspondence.

Theorem 4 (Completeness of *CLnS* Relative to *CLS*).

1. $\chi\{\xi_3\{1\}\} \xrightarrow{CLS} \chi\{\xi_1\{a\} \parr \xi_2\{\overline{a}\}\}$ *iff* $\chi\{\xi_3\{1\}\} \xrightarrow{CLnS} \chi\{\xi_1\{a\} \parr \xi_2\{\overline{a}\}\}$.
2. *If* A *and* B *are not dual literals, then* $\chi\{\xi_3\{A \parr B\}\} \xrightarrow{CLS} \chi\{\xi_1\{A\} \parr \xi_2\{B\}\}$ *iff* $\chi\{\xi_3\{A \parr B\}\} \xrightarrow{CLnS} \chi\{\xi_1\{A\} \parr \xi_2\{B\}\}$.

Proof. Every *CLnS* proof is manifestly a *CLS* proof under the interpretation of $*$ as $\parr$ and eliding the links. In the other direction, it suffices to observe that the lnk rule can be applied in such a way that the linked subformulas are the immediate operands of the principal $\parr$. Thus, every *CLS* rule that involves an interaction of two $\parr$-joined formulas can be simulated by a lnk, the corresponding *CLn* rule, and then unlnk or lnid. Once the full *CLS* derivation has been simulated in *CLnS* in this way, we remove all but the deepest linkages on A and B, and the corresponding instances of unlnk or lnid. □

Corollary 5. *A is provable in CLS if and only if it is provable in CLnS.*

Proof. *CLnS* lacks all the *CLS* rules that permute a $\parr$ into a smaller context: lsp, rsp, bng, add, and all. Thus, the only way for two $\parr$-joined formulas to interact is by means lnk followed by *CLn* rules. We can thus appeal to Thm. 4. □

The *CLn* rules preserve the number of occurrences of linkages and interactions. Moreover, they are biased to permute the negative connectives (*i.e.*, & and $\forall$) out of the positive connectives. This guarantees that if the conclusion of any *CLn* rule is provable in *CLnS*, then so is its premise; in other words, every rule of *CLn* is invertible. The order of application of the *CLn* rules is therefore immaterial. While different orders can produce different premises, the premises are equivalent.

Theorem 6 (Order Independence of *CLn*). *If* $A \xrightarrow{CLn} B$ *and* $A' \xrightarrow{CLn} B$, *then* $A \equiv A'$.[1]

Proof. Straightforward inspection of the rules of *CLn*. □

As a consequence, the only choices the user needs to explicitly instantiate an $\exists$ using wit, and contract a ?-formula using drl. The other choices in a proof—the multiplicative splits for $\otimes$ and disjunctive choices for $\oplus$—are *deterministically* inferred directly from the links.

[1] $A \equiv B$ can be defined as usual as $(\overline{A} \parr B) \,\&\, (\overline{B} \parr A)$.

Example 7. The following *CLnS* proof illustrates the use of linking.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall y, z.\, 1}{\forall y, z.\, \mathrm{u}{::}\overline{p(y)} * \mathrm{u}{::}p(y)}\,\mathsf{lnid}}{\forall y, z.\, (\mathrm{u}{::}\overline{p(y)} \oplus p(z)) * (\overline{p(c)} \oplus \mathrm{u}{::}p(y))}\,\mathsf{chc}\times 2}{\forall y.\, \big(\forall z.\, \mathrm{u}{::}\overline{p(y)} \oplus p(z)\big) * \big(\overline{p(c)} \oplus \mathrm{u}{::}p(y)\big)}\,\mathsf{lnall}}{\forall y.\, \big(\exists x.\, \forall z.\, \mathrm{u}{::}\overline{p(x)} \oplus p(z)\big) * \big(\overline{p(c)} \oplus \mathrm{u}{::}p(y)\big)}\,\mathsf{wit}}{\big(\exists x.\, \forall y.\, \mathrm{u}{::}\overline{p(x)} \oplus p(y)\big) * \big(\forall y.\, \overline{p(c)} \oplus \mathrm{u}{::}p(y)\big)}\,\mathsf{lnall}}{\big(\exists x.\, \forall y.\, \mathrm{u}{::}\overline{p(x)} \oplus p(y)\big) * \big(\exists x.\, \forall y.\, \overline{p(x)} \oplus \mathrm{u}{::}p(y)\big)}\,\mathsf{wit}}{\big(\exists x.\, \forall y.\, \overline{p(x)} \oplus p(y)\big) \parr \big(\exists x.\, \forall y.\, \overline{p(x)} \oplus p(y)\big)}\,\mathsf{lnk}$$

Observe that the *CLn* rules are interleaved with the other rules of *CLnS*.

It is instructive to compare the *CLn* rules with the similar *focused interaction* rules of *LSF* [7], which is a polarized and focused variant of *CLS*. Most of the rules for positive connectives are the same, except for lnbng,[2] which differs only in the fact that the interaction $*$ does not dissipate into a $\parr$ in *CLn*. As a result, *CLn* requires rules for interactions between negative formulas as well, while *LSF* defines focused interactions only for one positive interacting with one negative formula. Conceptually, a *CLnS* proof can be viewed as an *LSF* proof where several neighbouring (focused) interactions are merged into one.

3.2 Positive Equality and First-Order Effects

Let us call a sequence of applications of *CLnS* rules that begin at the bottom with lnk and end at the top with its corresponding unlnk or lnid a *linking phase*. Of the non-*CLn* rules in Ex. 7, the instances of drl and chc are all forced and can be done automatically. The instances of wit, on the other hand, require user guidance. The quantifier structure of this theorem requires the application of at least one lnall before a wit, so in the general case the system will not be able to keep all the rules of a single linking phase together in a block, *i.e.*, the linking phase are not *atomic*. Notice that this problem is only significant in the first-order case; for the propositional connectives, we can add rules such as the following to perform a non-*CLn* rule in the middle of a *CLn* derivation.

$$\dfrac{(\chi\{\mathrm{u}{::}A\}) * \xi\{\mathrm{u}{::}B\}}{(?\chi\{\mathrm{u}{::}A\}) * \xi\{\mathrm{u}{::}B\}}\,\mathsf{lndrl} \qquad \dfrac{\chi\{\mathrm{u}{::}A\} * \xi\{\mathrm{u}{::}C\}}{(\chi\{\mathrm{u}{::}A\} \oplus B) * \xi\{\mathrm{u}{::}C\}}\,\mathsf{lnlch} \qquad \dfrac{\chi\{\mathrm{u}{::}B\} * \xi\{\mathrm{u}{::}C\}}{(A \oplus \chi\{\mathrm{u}{::}B\}) * \xi\{\mathrm{u}{::}C\}}\,\mathsf{lnrch}$$

To make the linking phase atomic, we add an equality predicate to the language, change lnid to introduce equations between the terms, and add the following lnwit rule to *CLn*.

$$\dfrac{s_1 = t_1 \otimes \cdots \otimes s_n = t_n}{\mathrm{u}{::}p(s_1, \ldots, s_n) * \mathrm{u}{::}\overline{p(t_1, \ldots, t_n)}}\,\mathsf{lnid} \qquad \dfrac{\exists x.\, \big(\chi\{\mathrm{u}{::}A\} * \xi\{\mathrm{u}{::}B\}\big)}{\big(\exists x.\, \chi\{\mathrm{u}{::}A\}\big) * \xi\{\mathrm{u}{::}B\}}\,\mathsf{lnwit}$$

[2] In the polarized setting, there would be a similar case for the $\downarrow$ connective that can be seen as a purely linear variant of !.

Example 8. Let us revisit Ex. 7 using the above rules. We have this derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists x.\forall y.\exists x_1.\forall y_1.\, x_1 = y}{\exists x.\forall y.\exists x_1.\forall y_1.\, \mathfrak{u}{::}\overline{p(x_1)} * \mathfrak{u}{::}p(y)}\,\mathsf{Inid}}{\exists x.\forall y.\exists x_1.\forall y_1.\, (\mathfrak{u}{::}\overline{p(x_1)} \oplus p(y_1)) * (\overline{p(x)} \oplus \mathfrak{u}{::}p(y))}\,\mathsf{Inlch},\mathsf{Inrch}}{\exists x.\forall y.\exists x_1.\, \big(\forall z.\, \mathfrak{u}{::}\overline{p(y)} \oplus p(y_1)\big) * \big(\overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inall}}{\exists x.\forall y.\, \big(\exists x_1.\forall y_1.\, \mathfrak{u}{::}\overline{p(x_1)} \oplus p(y_1)\big) * \big(\overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inwit}}{\exists x.\, \big(\exists x_1.\forall y.\, \mathfrak{u}{::}\overline{p(x_1)} \oplus p(y)\big) * \big(\forall y.\, \overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inall}}{\big(\exists x.\forall y.\, \mathfrak{u}{::}\overline{p(x)} \oplus p(y)\big) * \big(\exists x.\forall y.\, \overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inwit}}{\big(\exists x.\forall y.\, \overline{p(x)} \oplus p(y)\big) \parr \big(\exists x.\forall y.\, \overline{p(x)} \oplus p(y)\big)}\,\mathsf{Ink}$$

The derivation can now keep all the *CLn* rules together.

A linking phase now ends (reading from conclusion to premise) with either Inid or unlnk. In Ex. 8 above, the residual premise after the Inid is essentially a first-order unification problem. The equations formed by the instances of Inid are all positively signed atomic formulas, with the following two rules.

$$\dfrac{1}{x = x}\,\mathsf{refl} \qquad \dfrac{s_1 = s_2 \otimes \cdots \otimes s_n = t_n}{f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)}\,\mathsf{cong}$$

Fact 9. *The rules* refl, cong, *and* $1 \longrightarrow 1 \otimes 1$ *suffice to show that* $t = t$ *for any term* t. □

Since every derivation can be embedded in any formula context, we thus obtain a smooth continuum between deduction and unification in terms of the equality predicate. This freedom comes at a price: the order of the applications of Inwit and Inall is now important.

Example 10. This is what happens if the Inwit and Inall rules of Ex. 8 are done in the wrong order.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists x, x_1.\forall y, y_1.\, x_1 = y_1}{\exists x, x_1.\forall y, y_1.\, \mathfrak{u}{::}\overline{p(x_1)} * \mathfrak{u}{::}p(y_1)}\,\mathsf{Inid}}{\exists x, x_1.\forall y, y_1.\, \big(\mathfrak{u}{::}\overline{p(x_1)} \oplus p(y)\big) * \big(\overline{p(x)} \oplus \mathfrak{u}{::}p(y_1)\big)}\,\mathsf{Inlch},\mathsf{Inrch}}{\exists x, x_1.\forall y.\, \big(\mathfrak{u}{::}\overline{p(x_1)} \oplus p(y)\big) * \big(\forall y_1.\, \overline{p(x)} \oplus \mathfrak{u}{::}p(y_1)\big)}\,\mathsf{Inall}}{\exists x, x_1.\, \big(\forall y.\, \mathfrak{u}{::}\overline{p(x_1)} \oplus p(y)\big) * \big(\forall y.\, \overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inall}}{\exists x.\, \big(\exists x_1.\forall y.\, \mathfrak{u}{::}\overline{p(x_1)} \oplus p(y)\big) * \big(\forall y.\, \overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inwit}}{\big(\exists x.\forall y.\, \mathfrak{u}{::}\overline{p(x)} \oplus p(y)\big) * \big(\exists x.\forall y.\, \overline{p(x)} \oplus \mathfrak{u}{::}p(y)\big)}\,\mathsf{Inwit}}{\big(\exists x.\forall y.\, \overline{p(x)} \oplus p(y)\big) \parr \big(\exists x.\forall y.\, \overline{p(x)} \oplus p(y)\big)}\,\mathsf{Ink}$$

The residual premise is unprovable.

It is not possible to fix this problem without further user guidance, because determining the correct quantifier nesting is equivalent to the full theorem proving problem for first-order logic, which is unsolvable. Fortunately, there is only a single critical pair that needs to be resolved: an interaction between two ∃-formulas. In every other case, we can appeal to Thm 6, suitably generalized.

Our solution therefore involves a modification to linkages. We now have two kinds of *oriented* linkages, $\mathsf{u}{\gg}A$ (called a *source*) and $\mathsf{u}{\ll}A$ (called a *sink*), with $\mathsf{u}{::}A$ standing for either form when irrelevant. The lnk rule is modified to produce a single source and sink pair:

$$\frac{\xi\{\chi_1\{\mathsf{u}{\gg}A\} * \chi_2\{\mathsf{u}{\ll}B\}\}}{\xi\{\chi_1\{A\} \parr \chi_2\{B\}\}}\,\mathsf{lnk}$$

Most of the remaining rules stay the same, except for the following new rule for resolving an interaction between $\exists$-formulas.

$$\frac{\exists y.\,\big((\exists x.\,\chi\{\mathsf{u}{\gg}A\}) * \xi\{\mathsf{u}{\ll}B\}\big)}{\big(\exists x.\,\chi\{\mathsf{u}{\gg}A\}\big) * \big(\exists y.\,\xi\{\mathsf{u}{\ll}B\}\big)}\,\mathsf{lnex}$$

Intuitively, the source is "brought to" the sink. The rules are prioritized so that the lnwit rule is only applicable in the case that lnex is not. Note that if the source is not on the left of the interaction $*$, then the lncom rule can be used to put it there. Let *CLonS* stand for:

$$\mathit{CLnS} \cup \{\mathsf{lndrl}, \mathsf{lnlch}, \mathsf{lnrch}, \mathsf{lnwit}, \mathsf{lnex}\} \cup \{\mathsf{refl}, \mathsf{cong}\}$$

(with the understanding that the lnk rule introduces oriented linkages), and let *CLon* stand for $\mathit{CLonS} \setminus \mathit{CLS}$.

Theorem 11 (Order Independence of *CLon*). *If* $A \xrightarrow{\mathit{CLon}} B$ *and* $A' \xrightarrow{\mathit{CLon}} B$, *then* $A \equiv A'$.

Proof. Same idea as for Thm. 6. □

Theorem 12. *For any formula* A *free of* $=$*-subformulas,* $1 \xrightarrow{\mathit{CLonS}} A$ *if and only if* $1 \xrightarrow{\mathit{CLnS}} A$.

Proof (Sketch). For the forward direction (only if), the *CLonS* rules lnwit, lnex, lndrl, lnlch and lnrch are admissible in *CLnS*. To recover the *CLnS* proof, it suffices to use the wit, drl and chc rules to rewrite all the instances of lnwit, lnex, lndrl, lnlch and lnrch. Then, these rules can be permuted below any occurrences of (the *CLonS* variant of) lnid on subformulas. This leaves just the instances of refl and cong above the *CLonS* variants of lnid; but, these rules do not interact with any other rules, and can therefore be permuted to stand in a block right above the lnid that gives rise to them; the entire block can then be replaced with the *CLnS* variant of lnid.

The reverse direction (if) is a simple consequence of Fact 9. □

4 Implementation Notes

The classical system *CLonS* has been implemented as part of the interactive proving tool *Profound* [6]. The tool is launched by giving it a goal theorem either on the command line or using an input file. It then presents the user with an interface where the text of the theorem may be directly manipulated (using the

keyboard or the mouse) in two *modes*: a *logical* mode and a *linking* mode. In the logical mode, the user is allowed to perform any of the rules in *CLonS* \ *CLon*. In other words, the only non-trivial actions available in this mode are: (1) instantiating $\exists$ (wit) and choice (chc) when there are no linkages, and (2) contraction (con) and weakening (wk). The user is also allowed the freedom to freely reorganize the formula using associativity and commutativity of $⅋$. The system lets the user also treat the other binary connectives as associative and commutative (the corresponding rules are admissible in *CLS* and *CLnS*).

In the linking mode, the user indicates an oriented link between a pair of subformulas. Using the keyboard, this amounts to traversing the subformula tree and marking two subformulas that are ancestrally $⅋$-linked as source and sink. This action is more intuitive with a pointing device such as the mouse, where one subformula is simply dragged and dropped on another. The main difficulty with pointing is to determine which subformulas the user wishes to point to, as most cursor positions occur within many subformulas. The system uses the heuristic of selecting the smallest subformula surrounding the cursor position, and letting the user go up in the subformula tree by scrolling the mouse-wheel.[3]

The current implementation of *Profound* is in OCaml and uses the GTK+ windowing library for its graphical interface, and has been tested to run on POSIX-compatible systems such as Linux and Mac OS X. In the future, we will probably re-implement it as a server/client setup using Javascript and using the HTML5 Canvas library for the user interface, which would make it portable across a wide range of platforms that support graphical web-browsers. Systems that can support multiple pointers and different interaction metaphors such as "pinching" might also enable *Profound* to perform marking or linking of formulas more directly.

In the rest of this section, we will describe some additional features of *Profound* that are not strictly part of the *CLonS* proof system, but are nevertheless very important for practical use of linking-based interaction.

4.1 Implicit Contraction

Perhaps the most natural aspect of the linking-based interaction metaphor is the ability to treat a logical problem as a kind of matching puzzle: bits of the current obligation are linked to bits of known facts and lemmas until the problem reduces to a known and manageable form. However, the formal *CLonS* system does require an additional step of explicit contraction that can complicates this simple aspect. For example, suppose we want to prove $?(a \otimes \overline{b}) ⅋ ?(a \otimes \overline{c}) ⅋ ?\overline{a} ⅋ (b \otimes c)$. The ideal linking-based proof would just link the $\overline{a}$ to the two occurrences of a. However, the first link would "consume" the $?\overline{a}$. The user is forced to explicitly contract it before drawing the links.

[3] Unfortunately, this turns out to be fairly frustrating when trying to select, for instance, the subformula $B ⅋ C$ from $A ⅋ B ⅋ C$, because $⅋$ is internally treated as a left-associative binary operator. The user is forced to first bring the entire subformula they wish to select to the left (which can be done freely in the logical mode).

In *Profound*, this issue is solved by means of special variants of lnk that store a copy of the ?formula for reuse in subsequent links. One such rule is as follows.

$$\frac{\xi\{?\chi_1\{A\} \parr (?\chi_1\{\mathrm{u}\gg A\} * \chi_2\{\mathrm{u}\ll B\})\}}{\xi\{?\chi_1\{A\} \parr \chi_2\{B\}\}}$$

Of course, this incurs the dual overhead of ?-formulas that survive unnecessarily. The user will need to clear them with explicit instances of wk, which can quickly become burdensome. From our experimentation, there does not seem to be a preferable default, so *Profound* allows both contracting and non-contracting uses of lnk. Indeed, *Profound* allows the user to independently decide to contract the source and the sink when marking them.

There is a related issue when marking the source and the sink that are not ancestrally $\parr$-related, but that have an ancestral ?. The simplest example is $?(a \oplus \overline{a})$. With the *CLonS* calculus and implicit linking as described, the user will still not be able to link the a and the $\overline{a}$ without first explicitly contracting the formula. This is not an altogether contrived example: consider the actual representation of the classical Drinker's formula, $?\exists x.\, \forall y.\, (p(x) \oplus \overline{p(y)})$, as opposed to its linear version in Ex. 7. To solve this issue, we further extend the lnk rule to allow contraction at an ancestral ? when there is no ancestral $\parr$ as follows.

$$\frac{\xi\{?(\chi_1\{\mathrm{u}\gg A\} \circ \chi_2\{B\}) * ?(\chi_1\{A\} \circ \chi_2\{\mathrm{u}\ll B\})\}}{\xi\{?(\chi_1\{A\} \circ \chi_2\{B\})\}}$$

where $\circ \in \{\otimes, \oplus, \&\}$.

4.2 Other Convenience Features

The *CLonS* system is cut-free, but in practice it is invaluable to use cuts in a proof, both for conceptual and textual simplicity. The *Profound* implementation therefore allows the user to introduce a cut at any moment, even in the middle of constructing a link. Instead of the general cut rule, however, we actually implement the following variant, which is a composition of the usual cut rule and rsp.

$$\frac{A \otimes (\overline{A} \parr C)}{C}$$

The rules for conjunctive truth on the last line of Fig. 2, while sufficient for completeness, are generally too cumbersome to use in practice. Instead, *Profound* actually implements the following monoidal versions (which are all derivable);

$$\frac{A}{A \circ \dagger} \qquad \frac{A}{\dagger \circ A} \qquad \frac{1}{1 \circ 1} \qquad \frac{A}{\forall x.\, A} \qquad \frac{A}{\exists x.\, A}$$

where $\langle \circ, \dagger \rangle \in \{\langle \otimes, 1 \rangle, \langle \oplus, 0 \rangle, \langle \&, \top \rangle\}$, and in the final two rules x is not free in A. In fact, these rules are not explicitly presented to the user, but are instead folded into the traversal mechanism. That is, whenever the cursor on $\dagger$ in $A \circ \dagger$ or $\dagger \circ A$, traversing to the parent simply changes the entire formula to A. This

allows us to implement chc in a very simple manner: we just add the following general rule:

$$\frac{\mathsf{0}}{A}\,\mathsf{del}$$

The user can use del to change any subformula to 0, and if it is the operand of a $\oplus$ then traversing to the parent will remove the 0 as a side effect. We find this to be a solid improvement over a direct use of chc, which would otherwise require a further clarification from the user.

5 Caveats

It is important to point out that *Profound* is only a research prototype at this point. To make it more broadly viable, we need at least two more crucial features that would require basic research on the calculus of structures.

- Support for intensional equality and induction: in order to make *Profound* suitable as a user interface for reasoning about computational specifications, the logic must be extended to support intensional (predicate) equality and induction. Equality can be supported by the rules of unification logic, which can be turned into a contextual and incremental form without too much effort. However, for practical uses it will need to support reasoning about equality transitively (*e.g.*, to show $\big((x \neq z) \otimes (x \neq s(z))\big) \parr \mathsf{0}$ where z and s represent zero and successor, respectively). In the general case this will amount to (incremental) congruence-closure.

  The general induction rules in the style of *LINC* [18] or μ*MALL* [2] can be readily added to the deep inference formalism. However, experience suggests that proofs written using such induction rules tend to be verbose and confusing. It is more standard to use more restricted induction schemas based on subterm or lexicographic ordering, such as in the Abella system [9]. Unfortunately, such restricted schemas tend to have a global and shallow flavour that runs counter to the incremental nature of deep inference. However, the final word is far from written on this matter.
- Support for typed and higher-order reasoning: supporting typed first-order terms is completely straightforward and *Profound* already assumes that the predicates and terms are simply typed. We intend to extend it to support polymorphically typed predicates and terms in the near future. Supporting dependently typed terms is not particularly critical as dependent restrictions can be recovered relationally; nevertheless, constructing a variant of the calculus of structure for dependent types is an open problem. Second-order quantification [16] should be straightforward. Extending the calculus of structures to full higher-order logic is also open.

Acknowledgements. We thank Dale Miller and Lutz Straßburger for their help with many aspects of this work.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2(3), 297–347 (1992)
2. Baelde, D.: Least and greatest fixed points in linear logic. ACM Trans. on Computational Logic 13(1) (April 2012)
3. Bertot, Y.: The CtCoq system: Design and architecture. Formal Aspects of Computing 11(3), 225–243 (1999)
4. Bertot, Y., Kahn, G., Théry, L.: Proof by pointing. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 141–160. Springer, Heidelberg (1994)
5. Boespflug, M., Carbonneaux, Q., Hermant, O.: The $\lambda\Pi$-calculus modulo as a universal proof language. In: Pichardie, D., Weber, T. (eds.) Proceedings of PxTP 2012: Proof Exchange for Theorem Proving, pp. 28–43 (2012)
6. Chaudhuri, K.: Profound (2013), `http://chaudhuri.info/software/profound/`
7. Chaudhuri, K., Guenot, N., Straßburger, L.: The Focused Calculus of Structures. In: Computer Science Logic: 20th Annual Conference of the EACSL, Leibniz International Proceedings in Informatics (LIPIcs), pp. 159–173. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (September 2011)
8. Chaudhuri, K., Pfenning, F., Price, G.: A logical characterization of forward and backward chaining in the inverse method. J. of Automated Reasoning 40(2-3), 133–177 (2008)
9. Gacek, A.: The Abella system and homepage (2009), `http://abella.cs.umn.edu/`
10. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
11. Thery, G.K.L., Bertot, Y.: Real theorem provers deserve real user-interfaces. In: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments. Software Engineering Notes, vol. 17(5). ACM Press (1992)
12. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science 410(46), 4747–4768 (2009)
13. McLaughlin, S., Pfenning, F.: Imogen: Focusing the polarized focused inverse method for intuitionistic propositional logic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 174–181. Springer, Heidelberg (2008)
14. Snow, Z., Baelde, D., Nadathur, G.: A meta-programming approach to realizing dependently typed logic programming. In: ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), pp. 187–198 (2010)
15. Straßburger, L.: Linear Logic and Noncommutativity in the Calculus of Structures. PhD thesis, Technische Universität Dresden (2003)
16. Straßburger, L.: Some observations on the proof theory of second order propositional multiplicative linear logic. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 309–324. Springer, Heidelberg (2009)
17. Stump, A.: Proof checking technology for satisfiability modulo theories. In: Abel, A., Urban, C. (eds.) Logical Frameworks and Meta-Languages: Theory and Practice (2008)
18. Tiu, A.: A Logical Framework for Reasoning about Logical Specifications. PhD thesis, Pennsylvania State University (May 2004)
19. Troelstra, A.S.: Lectures on Linear Logic. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California (1992)

Automatically Generated Infrastructure for De Bruijn Syntaxes

Emmanuel Polonowski

LACL, University Paris-East Créteil,
61 Avenue du Général de Gaulle, 94017 Créteil, France
emmanuel.polonowski@u-pec.fr
http://www.lacl.fr/~polonowski

Abstract. Dealing with variable binding during the formalization of programming languages metatheory is notorious for being a very complex issue. This paper introduces a new framework, *DBEB*, and a tool based on it, DBGen, whose purpose is to generate Coq code providing a rather complete infrastructure for de Bruijn encodings of a large variety of languages. *DBEB* is an abstract syntax with explicit binding informations that captures the great regularity of de Bruijn syntaxes. From this abstract syntax it is then possible to derive all the definitions and property statements and proofs required for the formalization of the syntactic infrastructure of the language.

Thereby, from a Coq inductive definition of a syntax in de Bruijn style, annotated with comments that make explicit its binding structure within *DBEB*, DBGen produces a Coq module with term structures definitions and a significant amount of properties (and their proof), up to the *substitution lemma*. Mutually defined syntaxes are supported, and such definitions may contain several distinct sets of variables. Moreover, this framework handles the generation of a named syntax for "usual" binding with explicit variables together with a smart translation function that greatly improves the readability of de Bruijn terms.

Keywords: De Bruijn syntax, formalization, infrastructure generation, Coq proof assistant.

1 Introduction

The large amount of work done in the last decade around the issue of variable binding in the mechanical formalization of programming languages metatheory highlights its importance and its complexity. Indeed, there is nowadays a wide choice of proof assistants, and of means to address this issue, each of them having their own strenghts and drawbacks. If we restrict ourselves to first-order encodings, the price of binders formalization is quite always to be paid, either by changing the formal foundations of term representations, or by requiring a sophisticated encoding of the original syntax or by a increasing the complexity of the whole metatheory.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 402–417, 2013.

Historically, the naive approach which consisted of using named variables in the minds of pencil-and-paper proofs has long since been abandoned because of the exorbitant management cost of α-conversion (*i.e.* irrelevance of bound variable names), and syntax encodings using de Bruijn indices [1] has then be preferred, even with the additional burden of arithmetic reasoning and the loss of readability it induced. From that starting point, many works have been done to address this issue, at the theoretical or at the practical level. To cite the most relevant w.r.t. the work presented here, we focus on three of them (for a more complete state of the art see [2] for instance): at the beginning of this century, Gabbay and Pitts [3,4] set the foundations of a new approach based on a slightly different basis for terms representation, which gave rise to the *nominal* framework [5]; More recently, Aydemir, Charguéraud at al. [2,6] proposed an interesting solution based on the *locally nameless* representation of terms by taking into account the difference between free and bound variables in the term structure; based on this approach, Aydemir and Weirich [7] developed the tool *LNgen* that generates locally nameless representations and infrastructure for some term grammars – this tool is probably the most proximate work to ours.

By contrast to the first two works, our contribution is indeed a very pragmatic one as it does not intend to address the theoretical issue of variable binding at the meta-level of programming languages theories, but rather aims at developping a tool that alleviates the burden of binding formalization and allows the user to concentrate on the interesting parts of its theories, while providing many infrastructure facilities. To achieve this goal, we resolutely choose to work with de Bruijn encodings, for several reasons. First, it is a well-known framework in the community, and it has clear merits since it is close to implementation mechanisms (abstract machines, etc.), completely first-order and thus very relevant to use with any proof assistant, there is furthermore an abundant literature that makes use of it. Second, despite the charges of opacity, it is quite close to pencil-and-paper: no complex binding structures, less adequacy problems (*i.e.* correctness of the encoding w.r.t. the informal specification), only standard meta-level foundations. Third, proof assistants are usually very efficient in their arithmetic reasoning capabilities, and automation can be widely used to deal with the extra arithmetic facts induced by de Bruijn encodings. Fourth, de Bruijn representations are *very* regular, as well as the main functions of lifting and substitution, as well as the main properties about those functions and their proof. Thus, the key idea is to take advantage of that regularity and of the arithmetic strength of proof assistant to completely automatize the generation of the needed infrastructure for de Bruijn representations. Furthermore, this approach allows us to generate a named syntax and its translation function to the de Bruijn one, providing a handy way to write de Bruijn terms in the user work.

DBGen[1] [8] is a tool in the spirit of *LNgen*: it takes as input a grammar and produces definitions, lemmas, proofs and tactics for the Coq proof assistant. It has been successfully used to generate infrastructure for quite complex languages, involving mutually defined term structures and multiple variable sets.

[1] Available at `http://www.lacl.fr/~polonowski/Develop/DBGen/dbgen.html`

The generated code is well-organized in a hand-written style, allowing the user to browse through it and reuse parts of it as needed. It takes advantage of the module system of Coq: the infrastructure can be compiled only once, until a modification of the input grammar; and such modifications are easily managed since all the infrastructure can be immediately generated again. DBGen is developed in OCaml with a clear structure designed to be easily extended: functions and properties regular w.r.t. the de Bruijn representation can be added at a very small development cost.

The remainder of the paper is organized as follows: we recall in Section 2 how the de Bruijn setting eases the definitions of term structures and lifting and substitution operations; Section 3 introduces de Bruijn with Explicit Binding abstract syntax as a relevant framework that captures the regularity of de Bruijn structures needed for automatic generation; two examples of DBGen usage are detailed in Section 4; Section 5 concludes and gives hints to further work.

2 Usual de Bruijn Syntaxes and Their Infrastructure

De Bruijn syntaxes are nowadays well-known and do not require a detailed introduction here. Let us simply recall that, in his early work [1], de Bruijn introduced an alternative representation of λ-terms [9] which does not rely upon a dynamic binding with names (which comes with α-equivalence): it uses indices to statically link the place of a variable with its binder.

Despite the benefits of this approach, two flaws are commonly pointed out (see for instance POPLmark Challenge [10]): the loss of readability and the arithmetic additional work needed to deal with this binding mechanism. However, it has a very regular infrastructure which extends naturally to complex term structures involving several distinct variable sets, mutually defined syntactic categories and binding of several variables at a time.

Let us illustrate this with an example combining the first both difficulties: an extension of Girard's system $\mathcal{F}$ [11] with arithmetic expressions, noted $\mathcal{F}_{expr}$. The named and de Bruijn syntax definitions of $\mathcal{F}_{expr}$ are given as follows:

Named syntax	**De Bruijn syntax**
$A ::= X \mid A \to A \mid \Pi X.A$	$A ::= \bar{X} \mid A \to A \mid \Pi A$
$t ::= \lambda x : A.t \mid t\ t \mid t\ A \mid \Lambda X.t \mid e$	$t ::= \lambda A.t \mid t\ t \mid t\ A \mid \Lambda t \mid e$
$e ::= x \mid n \mid e + e \mid t$	$e ::= \bar{x} \mid n \mid e + e \mid t$

Note that in usual de Bruijn grammar there is no formal way to know which construction is a binder, and, for those ones, which index category is bound. In a pencil-and-paper work, one may leave to the reader the formal definition of the substitution functions, discarding the tedious details of the mutual definitions induced by this grammar. Figure 1 and 2 give the named substitution definitions, we show it to make explicit the complexity of this task and exhibit some crucial points.

Let us point out the rule $(\Lambda Y.t)\{e/x\} = \Lambda Z.t\{Z/Y\}\{e/x\}$ which makes use of renaming for the bound variable Y because of its possible occurrence in e;

$$
\begin{array}{lll}
X\{B/X\} & = B & \\
Y\{B/X\} & = Y & \text{if } X \neq Y \\
(A_1 \to A_2)\{B/X\} & = A_1\{B/X\} \to A_2\{B/X\} & \\
(\Pi Y.A)\{B/X\} & = \Pi Z.A\{Z/Y\}\{B/X\} & \text{with } Z \text{ fresh}
\end{array}
$$

$$
\begin{array}{lll}
(\lambda y : A.t)\{B/X\} & = \lambda y : A\{B/X\}.t\{B/X\} & \\
(t_1\ t_2)\{B/X\} & = t_1\{B/X\}\ t_2\{B/X\} & \\
(t\ A)\{B/X\} & = t\{B/X\}\ A\{B/X\} & \\
(\Lambda Y.t)\{B/X\} & = \Lambda Z.t\{Z/Y\}\{B/X\} & \text{with } Z \text{ fresh}
\end{array}
$$

$$
\begin{array}{ll}
y\{B/X\} & = y \\
n\{B/X\} & = n \\
(e_1 + e_2)\{B/X\} & = e_1\{B/X\} + e_2\{B/X\}
\end{array}
$$

Fig. 1. $\mathcal{F}_{expr}$ type substitution functions

$$
\begin{array}{lll}
(\lambda y : A.t)\{e/x\} & = \lambda z : A.t\{z/y\}\{e/x\} & \text{with } z \text{ fresh} \\
(t_1\ t_2)\{e/x\} & = t_1\{e/x\}\ t_2\{e/x\} & \\
(t\ A)\{e/x\} & = t\{e/x\}\ A & \\
(\Lambda Y.t)\{e/x\} & = \Lambda Z.t\{Z/Y\}\{e/x\} & \text{with } Z \text{ fresh}
\end{array}
$$

$$
\begin{array}{lll}
x\{e/x\} & = e & \\
y\{e/x\} & = y & \text{if } x \neq y \\
n\{e/x\} & = n & \\
(e_1 + e_2)\{e/x\} & = e_1\{e/x\} + e_2\{e/x\} &
\end{array}
$$

Fig. 2. $\mathcal{F}_{expr}$ expression substitution functions

this case is quite tricky since three syntactic categories are involved, and the fact that type variables may be bound in expression is not obvious in usual de Bruijn syntax definition.

The syntax of this language in the de Bruijn setting leads to the definition of five lifting and five substitution functions... One can easily imagine the amount of work needed to perform those definitions, and furthermore to prove the infrastructure properties for all of them. Moreover, if errors happen to be in the language definition or if the syntax has to be modified, extended or shortened, all this work has to be modified and checked carefully. Automation can be very helpful for this issue, which is also one of the other approaches of binding formalization.

3 An Abstract Syntax for De Bruijn Infrastructure Generation

With the merits and the flaws of usual de Bruijn encodings in mind, we can now introduce a suitable framework for infrastructure formalization generation. We begin with the definition of an abstract syntax that captures the required

regularity of term structures, then we use it to generate lifting and substitution functions and properties and furthermore named syntax and translation.

3.1 De Bruijn with Explicit Binding Abstract Syntax

To proceed with automation, we need a clear and regular structure for such term definitions. The abstract syntax we propose here follows the tradition of abstract higher-order reduction systems (HORS, see [12] for a survey) with a main difference: where they propose a framework to define arbitrary reduction systems, we clearly restrict ours to a "standard" substitution mechanism in order to be able to generate the systems infrastructure.

Following the usual encodings of term structures in proof assistants, a syntactic category in the de Bruijn setting is defined by the mean of constructors of two kinds: index constructors and non-index constructors, the latter potentially having parameters (or sub-terms); those parameters are then given by a name and a type, and they can optionally bind in the subterm several variables of several distinct categories.

We introduce the de Bruijn with Explicit Binding abstract syntax (*DBEB*) as follows:

$$
\begin{array}{lll}
T, U ::= \bar{x} & & \text{Index variable constructor} \\
\quad \mid \; C\ p_1\ \ldots\ p_n & & \text{Ordinary constructor with parameters} \\
\\
p \quad ::= s : T & & \text{Ordinary parameter} \\
\quad \mid \; [n\ U]\ p & & \text{Parameter with binding in the subterm}
\end{array}
$$

The parameter with binding needs some attention because all the binding information needed for de Bruijn infrastructure generation depends on it. Between every brackets are given a natural number n_i and a syntactic category U_i, which says that the first n_i index of category U_i are bound in the subterm s of this parameter. For instance, the abstraction of ordinary λ-calculus $\lambda x.t$ (in a category named *term*) can be defined with this grammar as $\lambda\,([1\ \mathit{term}]\ \ t : \mathit{term})$.

Note that if we remove the binding information from the definition as *DBEB* we obtain the usual de Bruijn definition. Figure 3 gives the definition of $\mathcal{F}_{expr}$ as *DBEB*.

In addition to the explicit binding information, our syntax differs from usual higher-order systems by considering binders in constructor parameters instead of constructors themselves. It is of course possible to encode *DBEB* structures as HORS, but at the cost of splitting constructor parameters and defining them as new constructors, and the resulting term structure is then further away from the pencil-and-paper definition.

To conclude with this section, let us point out that *DBEB* serves only as a theoretical background for DBGen development but is not actually defined in Coq (neither in the content generated by DBGen). Such an approach of a meta-language with its own proofs and an encoding mechanism would certainly have its merits (see GMeta [13] for instance) but would also give rise to non-trivial adequacy issues whereas our choice is to stick to usual de Bruijn encoding.

$$
\begin{array}{lll}
A ::= & & type ::= \\
\quad | \;\; X & & \quad | \;\; \bar{X} \\
\quad | \;\; A \to A & & \quad | \;\; (A : type) \to (A : type) \\
\quad | \;\; \Pi X.A & & \quad | \;\; \Pi\,([1\ type]\, A : type) \\
\\
t ::= & & term ::= \\
\quad | \;\; \lambda x : A.t & & \quad | \;\; \lambda\,(A : type)\,([1\ expr]\ \ t : term) \\
\quad | \;\; t\ t & & \quad | \;\; (t : term)\ \ (t : term) \\
\quad | \;\; t\ A & & \quad | \;\; (t : term)\ \ (A : type) \\
\quad | \;\; \Lambda X.t & & \quad | \;\; \Lambda\,([1\ type]\ \ t : term) \\
\quad | \;\; e & & \quad | \;\; (e : expr) \\
\\
e ::= & & expr ::= \\
\quad | \;\; x & & \quad | \;\; \bar{x} \\
\quad | \;\; n & & \quad | \;\; (n : nat) \\
\quad | \;\; e + e & & \quad | \;\; (e : expr) + (e : expr) \\
\quad | \;\; t & & \quad | \;\; (t : term)
\end{array}
$$

Fig. 3. $\mathcal{F}_{expr}$ definition as *DBEB*

3.2 *DBEB* Infrastructure Generation

DBEB definitions provide all the needed information to deal with lifting and substitution functions generation, and moreover with basic infrastructure properties generation. Let us note that no specific induction schemes are needed to work with the languages we consider, all proofs proceeds by straightforward induction on the structure of terms.

We consider the following predicates over *DBEB* structures: *indexed* (T) stands if the syntactic category T does have an index constructor, *reach (T, U)* stands if *indexed* (T) stands and if there is a path from the syntactic category U to T, possibly through other categories, *via* their parameters; for instance, for $\mathcal{F}_{expr}$ the following statements hold: *indexed* $(type)$, *indexed* $(expr)$, *reach* $(type, term)$, *reach* $(type, expr)$, *reach* $(expr, term)$.

Indeed, the lifting and the substitution functions proceed with a great regularity over *DBEB* term structures. For each function, we indicate the name of the indexed category on which it works and the actual category which is processed. For instance in system $\mathcal{F}_{expr}$, the substitution function of type variable which goes through expressions will be noted $e\left\{B/\bar{X}\right\}_{type/expr}$ and the corresponding lifting function $e \uparrow^{n}_{m:type/expr}$.

Figure 4 presents the lifting function definition for an indexed category T in a processed category U, the key cases are those for a variable and a binder. Notice that crossing a binder of a distinct index category has no effect on the lifting operation.

For the same involved categories, we show in Figure 5 the substitution function definition. Let us point out the cases for a binder, where we make use of the lifting function in order to update the indices of the substituted term.

$$
\begin{array}{lll}
(\bar{x}) \uparrow^n_{m:T/U} & = \bar{x} & \text{if } T \neq U \\
(\bar{x}) \uparrow^n_{m:U/U} & = \overline{x+n} & \text{if } m \leqslant x \\
(\bar{x}) \uparrow^n_{m:U/U} & = \bar{x} & \text{if } x < m \\
(C\ p_1\ \ldots\ p_k) \uparrow^n_{m:T/U} & = C\ p_1 \uparrow^n_{m:T/U}\ \ldots\ p_k \uparrow^n_{m:T/U} & \\
 & & \\
(s : V) \uparrow^n_{m:T/U} & = s : V & \text{if } \neg \mathit{reach}\,(T, V) \\
(s : V) \uparrow^n_{m:T/U} & = s \uparrow^n_{m:T/V} : V & \text{if } \mathit{reach}\,(T, V) \\
([k\ T]\ p) \uparrow^n_{m:T/U} & = [k\ T]\ p \uparrow^n_{m+k:T/U} & \\
([k\ V]\ p) \uparrow^n_{m:T/U} & = [k\ V]\ p \uparrow^n_{m:T/U} & \text{if } T \neq V
\end{array}
$$

Fig. 4. Lifting in *DBEB*

$$
\begin{array}{lll}
(\bar{x})\,\{t/n\}_{T/U} & = \bar{x} & \text{if } T \neq U \\
(\bar{x})\,\{t/n\}_{U/U} & = \overline{x-1} & \text{if } n < x \\
(\bar{x})\,\{t/x\}_{U/U} & = t & \\
(\bar{x})\,\{t/n\}_{U/U} & = \bar{x} & \text{if } x < n \\
(C\ p_1\ \ldots\ p_k)\,\{t/n\}_{T/U} & = C\ p_1\,\{t/n\}_{T/U}\ \ldots\ p_k\,\{t/n\}_{T/U} & \\
 & & \\
(s : V)\,\{t/n\}_{T/U} & = s : V & \text{if } \neg \mathit{reach}\,(T, V) \\
(s : V)\,\{t/n\}_{T/U} & = s\,\{t/n\}_{T/V} : V & \text{if } \mathit{reach}\,(T, V) \\
([k\ T]\ p)\,\{t/n\}_{T/U} & = [k\ T]\ p\left\{t \uparrow^k_{0:T/T} /n+k\right\}_{T/U} & \\
([k\ V]\ p)\,\{t/n\}_{T/U} & = [k\ V]\ p\{t \uparrow^k_{0:V/T} /n\}_{T/U} & \text{if } T \neq V \text{ and } \mathit{reach}\,(V, T) \\
([k\ V]\ p)\,\{t/n\}_{T/U} & = [k\ V]\ p\,\{t/n\}_{T/U} & \text{if } T \neq V \text{ and } \neg \mathit{reach}\,(V, T)
\end{array}
$$

Fig. 5. Substitution in *DBEB*

Those two definitions are given inside *DBEB* but, for DBGen purpose, we will generate definitions in the usual de Bruijn syntax by removing the explicit binding and typing informations.

The next step is to generate properties about lifting and substitution. This will allow the DBGen user to concentrate on more important properties of the language he considers. Indeed, *DBEB* is enough to generate a lot of property statements and, moreover, it generates correct Coq proofs. This is due to the regularity of *DBEB* structures where all the proofs we consider proceeds solely by structural induction and let the complexity of the binding structure management be in arithmetic properties for which proof assistants are usually powerful. Since *DBEB* is not defined in the generated content, all the proofs are generated on a ad-hoc basis for the given language.

A first set of properties we need is about the independence of functions that deals with distinct indexed categories. For instance in system $\mathcal{F}_{expr}$, the lifting functions for type and expression variables can freely commute, this is also true for lifting and substitution functions. The second set of properties is more usual in de Bruijn infrastructure and talks about lifting simplification, lifting composition with itself, lifting composition with substitution and substitution composition with itself, also known as the *substitution lemma*. All those properties are listed in Figure 6.

- $t \uparrow^{n'}_{m':T/U} \uparrow^{n}_{m:V/U} = t \uparrow^{n}_{m:V/U} \uparrow^{n'}_{m':T/U}$
- $t\,\{u/p\}_{T/U} \uparrow^{n}_{m:V/U} = t \uparrow^{n}_{m:V/U} \{u \uparrow^{n}_{m:V/T} /p\}_{T/U}$
- $t \uparrow^{0}_{m:T/U} = t$
- $m' \leqslant m \leqslant n' + m' \Rightarrow t \uparrow^{n'}_{m':T/U} \uparrow^{n}_{m:T/U} = t \uparrow^{n'+n}_{m':T/U}$
- $n' + m' \leqslant m \Rightarrow t \uparrow^{n'}_{m':T/U} \uparrow^{n}_{m:T/U} = t \uparrow^{n}_{m-n':T/U} \uparrow^{n'}_{m':T/U}$
- $m \leqslant p < m + n \Rightarrow (t \uparrow^{n}_{m:T/U})\,\{u/p\}_{T/U} = t \uparrow^{n-1}_{m:T/U}$
- $0 < n \Rightarrow t \uparrow^{n}_{m+1:T/U} \{\bar{m}/m\}_{T/U} = t \uparrow^{n-1}_{m+1:T/U}$
- $m \leqslant p \Rightarrow t\,\{u/p\}_{T/U} \uparrow^{n}_{m:T/U} = t \uparrow_{m:T/U} \{u \uparrow_{m:T/T} /p+n\}_{T/U}$
- $p \leqslant m \Rightarrow t\,\{u/p\}_{T/U} \uparrow^{n}_{m:T/U} = t \uparrow^{n}_{m+1:T/U} \{u \uparrow^{n}_{m:T/T} /p\}_{T/U}$
- $m \leqslant n \Rightarrow t\,\{v/m\}_{T/U}\,\{u/n\}_{T/U} = t\{u \uparrow^{1}_{m:T/T} /n+1\}_{T/U}\{v\,\{u/n\}_{T/T} /m\}_{T/U}$

Fig. 6. Generic *DBEB* infrastructure statements

3.3 Named Syntax Generation and Translation

With *DBEB* it is very easy to define new syntaxes and functions and to generate them. We illustrate this with a named syntax and a translation function from it to the de Bruijn syntax, which are interesting enough themselves to be introduced here as they provide a way to greatly improve the readability of de Bruijn terms.

First, we take a type for named variables, say *name* (usual instances would be *string* or *nat*), and we only require a decidable equality over its values.Since we have binders of arbitrary arity in *DBEB*, we need a way to define named binders with the same capabilities, so we choose to bind over lists of variables. We assume the definition of length-indexed list of names and we shall note $names^k$ such a list of length k (this is easy to define in a proof assistant with dependent types, like Coq).

In addition to the constructions of the named syntax, we want to be able to embed a de Bruijn term inside a named term. We design this embedding to be smart enough to provide a way to use named terms as a way to write more clearly definitions and statements within the de Bruijn setting (we shall see an example in the next section), so we add to each syntactic category a new constructor named *DB* that takes as parameter the embedded de Bruijn term and a list of variable names capturing its first free indices.

We are now ready to generate the named syntax, as defined in Figure 7.

For instance, the generated named syntax for usual λ-calculus is the following grammar, where we note $"x"$ a named variable x and $t\,\lfloor "x_1", \ldots, "x_k" \rfloor$ the

$$
\begin{array}{ll}
\mathit{Named}(\bar{x})_T & = \underline{x} : \mathit{name} \\
\mathit{Named}(C\ p_1\ \ldots\ p_k)_T & = \underline{C}\ \mathit{Named}(p_1)_T\ \ \ldots\ \mathit{Named}(p_k)_T \\
 & \quad + \underline{DB}\ \ (k : \mathit{nat})\ \ \left(l : \mathit{names}^k\right)\ \ (t : T) \\
 & \\
\mathit{Named}(s : V)_T & = s : V \\
\mathit{Named}([k\ V]\ p)_T & = \left[l : \mathit{names}^k\right]\ \ \mathit{Named}(p)_T
\end{array}
$$

Fig. 7. Named syntax generation

special *DB* constructor for a given variable list $"x_1", \ldots, "x_k"$ and a de Bruijn term t:

$$\underline{t} ::= "x" \mid (\underline{t}\ \underline{u}) \mid \underline{\lambda}"x".\underline{t} \mid t\ \lfloor "x_1", \ldots, "x_k" \rfloor$$

Interesting examples of terms with this syntax involve the special *DB* constructor. For instance, the named term $\underline{\lambda}"x".\underline{\lambda}"y".t\ \lfloor "x", "y" \rfloor$ corresponds to the term t with its two first variables captured by $"x"$ and then by $"y"$, which is different from the term $\underline{\lambda}"x".\underline{\lambda}"y".t\ \lfloor "y", "x" \rfloor$. The named term $\underline{\lambda}"x".t\ \lfloor \rfloor$ is a function whose argument is useless since it does not capture any free variable of t; $\underline{\lambda}"x".(t\ \lfloor \rfloor\ "x")$ is even a more interesting term since it corresponds to the left hand side of the usual η-reduction rule where the bound variable $"x"$ must not be free in the left subterm.

To use this named syntax as a front-end for complex de Bruijn terms, we need a translation function that exactly captures the intuition of the previous examples. This is not a difficult task, but requires a little bit of attention since we potentially deal with several categories of variables. The idea of the translation function is standard: we collect the variable names along the traversal of the term, and when we arrive at a variable, we replace it by its position in the ordered collection; this strategy gives the usual account for name scopes.

More work has to be done with respect to the special constructor *DB* since it performs some non-trivial operation on the embedded de Bruijn term. Let us take as examples the following terms: $\underline{\lambda}"x".\underline{\lambda}"y".(\bar{0}\ \bar{1})\ \lfloor "y", "x" \rfloor$ must be translated to $\lambda\lambda(\bar{0}\ \bar{1})$ and $\underline{\lambda}"x".\underline{\lambda}"y".(\bar{0}\ \bar{1})\ \lfloor "x", "y" \rfloor$ must be translated to $\lambda\lambda(\bar{1}\ \bar{0})$, remark that some permutation of free indices are needed here. Moreover, free indices of the embedded term that are not to be captured need to be lifted and thus remain free.

To achieve this transformation, we need a function that is able to permute free indices of a de Bruijn term and also lift some of them. This function takes to parameters: k the amount of binders introduced upon the de Bruijn term, this will be used to perform the lifting; $[i_1, \ldots, i_n]$ the permutation given as a list of indices; m the number of local binders that have been crossed, since the bound variables of the embedded term must remain unchanged. Figure 8 gives the definition of this function on *DBEB* terms.

With the help of this function, we easily write the named to de Bruijn translation function. Some elementary operations on lists are used: dependent list concatenation (noted $\circ$), permutation definition from two ordered lists (noted &), list scanning (noted #); we refer the interested reader to the technical

$$
\begin{array}{lll}
(\bar{x})\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = \bar{x} & \text{if } x < m \\
(\bar{x})\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = \overline{x+k} & \text{if } m \leqslant x \text{ and } n \leqslant x-m \\
(\bar{x})\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = \overline{m+i_{x-m}} & \text{if } m \leqslant x \text{ and } x-m < n \\
(C\ p_1\ \ldots\ p_l)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = C\ (p_1)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]}\ \ \ldots\ \ (p_l)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & \\
 & & \\
(s : V)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = s : V & \text{if } \neg reach\,(T,V) \\
(s : V)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = s\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} : V & \text{if } reach\,(T,V) \\
([l\ T]\ p)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = [l\ T]\ p\updownarrow^{k,m+1,T}_{[i_1,\ldots,i_n]} & \\
([l\ V]\ p)\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & = [l\ V]\ p\updownarrow^{k,m,T}_{[i_1,\ldots,i_n]} & \text{if } T \neq V
\end{array}
$$

Fig. 8. Free indices permutation and lifting in *DBEB*

$$
\begin{array}{ll}
\underline{x}\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} & = \overline{l_{T_i}\#x} \\
(\underline{C}\ p_1 \ldots p_k)\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} & = C\ p_1\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i}\ \ \ldots\ p_k\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} \\
(\underline{DB}\ k\ l\ t)\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} & = t\updownarrow^{m_i-k,0,T_i}_{l_{T_i}\&l} \\
 & \\
(s : V)\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} & = s\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} : V \\
\left(\left[l' : names^k\right]\ p\right)\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i} & = [k\ T_i]\ p\left\langle l^{m_1}_{T_1},\ldots,(l'\circ l)^{k+m_i}_{T_i},\ldots,l^{m_n}_{T_n}\right\rangle^{T_i}
\end{array}
$$

Fig. 9. Named to de Bruijn translation function in *DBEB*

documentation for further details. The translation function takes as argument one dependent list of collected bound variable names per index category, together denoted $\left\langle l^{m_1}_{T_1},\ldots,l^{m_n}_{T_n}\right\rangle$, its definition is given in Figure 9.

To illustrate this definition, we give here the generated function for the named syntax given above for usual λ-calculus.

$$
\begin{array}{ll}
\underline{x}\left\langle l^m\right\rangle & = \overline{l\#x} \\
(\underline{t}\ \underline{u})\left\langle l^m\right\rangle & = t\left\langle l^m\right\rangle\ u\left\langle l^m\right\rangle \\
(t\left\lfloor "x_1",\ldots,"x_n"\right\rfloor)\left\langle l^m\right\rangle & = t\updownarrow^{m-n,0}_{l\&["x_1",\ldots,"x_n"]} \\
(\underline{\lambda}"x".\underline{t})\left\langle l^m\right\rangle & = \lambda t\left\langle "x" :: l^{m+1}\right\rangle
\end{array}
$$

With this function, we can translate our examples:

$$
\begin{aligned}
(\underline{\lambda}"x".\underline{\lambda}"y".(\bar{0}\ \bar{1})\left\lfloor "y","x"\right\rfloor)\left\langle\right\rangle &= \lambda\lambda\left((\bar{0}\ \bar{1})\left\lfloor "y","x"\right\rfloor\right)\left\langle ["y","x"]\right\rangle \\
&= \lambda\lambda\,(\bar{0}\ \bar{1})\updownarrow^{0,0}_{[0,1]} \\
&= \lambda\lambda\,(\bar{0}\ \bar{1})
\end{aligned}
$$

$$
\begin{aligned}
(\underline{\lambda}"x".\underline{\lambda}"y".(\bar{0}\ \bar{1})\left\lfloor "x","y"\right\rfloor)\left\langle\right\rangle &= \lambda\lambda\left((\bar{0}\ \bar{1})\left\lfloor "x","y"\right\rfloor\right)\left\langle ["y","x"]\right\rangle \\
&= \lambda\lambda\,(\bar{0}\ \bar{1})\updownarrow^{0,0}_{[1,0]} \\
&= \lambda\lambda\,(\bar{1}\ \bar{0})
\end{aligned}
$$

We also easily check that the translation of a term $\underline{\lambda}"x".(t \lfloor\rfloor\ "x")$, for any t, gives the term $\lambda\,(t'\ \bar{0})$ where t' is t with all its free indices lifted by 1; it is then exactly the left hand side of the η-reduction rule, which can thus be written $(\underline{\lambda}"x".(t \lfloor\rfloor\ "x"))\,\langle\rangle \to t$ without having to explicitely write the correct lifting operation at the left hand side.

To conclude, this translation function allows us to write de Bruijn terms as named terms with a fine notion of capture which proves to be enough to handle many definitions involving freshness. A detailed example of this usage is given in Section 4.2.

3.4 *DBEB* and DBGen Generation Questions

To conclude with the DBGen approach of generating content, even named syntaxes, from *DBEB* initial syntaxes (and not the contrary), we discuss here the limits and the future of this approach.

Can DBGen fail ? Syntactically invalid DBGen input will of course cause the tool to fail to produce any output, and presumably there are errors in the source code that might cause the generation to fail in unexpected circumtances... However the following question remains: Can DBGen fail to generate a valid (w.r.t. Coq) de Bruijn infrastructure from a valid *DBEB* syntax ? The answer we give is two-fold: no, we have not found any example that gives rise to an invalid generated content; perhaps, since the concept of *valid DBEB* should be more precisely described (a type system have recently been formalized for that purpose and integrated into DBGen v0.5.2).

Why using de Bruijn-like syntax as input ? Let us point out that it is of course possible to derive a *DBEB* structure from a named syntax provided that the necessary information is given as input (mainly the arity and the category of binders); Ott [14] would be a very good choice for that purpose, with a backend in the spirit of that for the locally-nameless representation of terms. However this would have two major flaws : first, the user would not be able to validate his de Bruijn definition before generating content over it, and we cannot (for now) ensure the correctness of DBGen output for incorrect inputs w.r.t. Coq inductive definitions; second, the user might consider that knowledge about de Bruijn structure is unnecessary while further uses of the generated content requires it (for now).

4 DBGen at Work

We focus now on the tool DBGen itself and we illustrate the usage of the generated infrastructure with two motivating example. The second one makes use of the tactics generated by DBGen that uses the infrastructure properties to simplify arbitrary terms with lifting and substitutions. This provides a very handy framework to prove further properties without the pain of knowing the details of

the de Bruijn infrastructure. This section requires some knowledge about inductive syntax definitions, we will use very simple Coq definitions and only some notations to facilitate the reading of the final example.

4.1 An Example of Generation

The language chosen as input is Coq itself as its knowledge is a requirement to be able to use the output of DBGen, the user gives his source syntax in the de Bruijn setting, and add comments to indicate where are the index constructors and the binders, *i.e.* describe it as a *DBEB* syntax.

The source syntax of λ-calculus with n-ary *let* and tuples is given as follows:

```
Module STLCletn.

  Inductive term : Type :=
  | var ((* index *) x : nat)
  | app (t1 : term) (t2 : term)
  | lam ((* bind term in *) t : term)
  | tuple (tl : terms)
  | letn (n : nat) (t : term) ((* bind [ n term ] in *) u : term)

  with terms : Type :=
  | tnil
  | tcons (t : term) (ts : terms).

End STLCletn.
```

DBGen output is around 1300 lines long, organized as a module named `STLCletn` (given in the source file) that contains the definition itself (without the comments) and the following lifting and substitution definitions (we present only the prototypes):

```
  Fixpoint term_lift_in_term (_n : nat) (_m : nat) (_arg : term) : term
  with term_lift_in_terms (_n : nat) (_m : nat) (_arg : terms) : terms.
  Fixpoint term_subst_in_term (_a : term) (_m : nat) (_arg : term) : term
  with term_subst_in_terms (_a : term) (_m : nat) (_arg : terms) : terms.
```

The named syntax is also automatically defined (where `_name_list` is the type of dependent lists of names), along with the translation function:

```
  Inductive _term : Type :=
  | _var (x : string)
  | _app (t1 : _term) (t2 : _term)
  | _lam (xl : _name_list 1) (t : _term)
  | _tuple (tl : _terms)
  | _letn (n : nat) (t : _term) (xl : _name_list n) (u : _term)
  | _db_term (_xn : nat) (_xl_term : _name_list _xn) (_arg : term)

  with _terms : Type :=
```

```
| _tnil
| _tcons (t : _term) (ts : _terms)
| _db_terms (_xn : nat) (_xl_term : _name_list _xn) (_arg : terms).

Fixpoint named_to_db_term
           (_n : nat) (_l : _name_list _n) (_arg : _term) : term
with named_to_db_terms
           (_n : nat) (_l : _name_list _n) (_arg : _terms) : terms.
```

Other examples are provided in the DBGen distribution, among them is the language $\mathcal{F}_{\text{expr}}$ presented before (DBGen output is around 3200 lines long) and also the even more complex language $\mathbf{Loop}^{\omega}$ [15,16] (6 syntactic categories with 2 of them having index constructors; DBGen output is around 5300 lines long).

4.2 An Example of Generated Content Usage

Let us take as concluding example the definitions of the abbreviations **val** and **let val** for the continuation monad in the λ-calculus (see [15] for more details about this example). Notice that we have here simplified the example for the sake of readability (and of character encodings) – the complete and Coq-checked example is available with the DBGen distribution[2].

The pencil-and-paper definitions are the following, where z is a *fresh* variable:

$$\mathbf{val}\ u = \lambda z.\,(z\ u)$$
$$\mathbf{let\ val}\ x = u\ \mathbf{in}\ t = \lambda z.\,(u\ \lambda x.\,(t\ z))$$

The corresponding definitions in the de Bruijn setting are quite complex, they require the use of the lifting function to preserve the binding structure with the addition of the new binder λz. An important property to establish for those abbreviations is the following.

Proposition 1. *For any terms t and u,*

$$\mathbf{let\ val}\ x = \mathbf{val}\ u\ \mathbf{in}\ t = t\,\{u/x\}$$

Proof. We **expand the abbreviations**, and get for the left hand side of the equation

$$\lambda z.\,((\lambda z'.\,(z'\ u))\ \ \lambda x.\,(t\ z))$$

The proof the proceeds in 3 steps:

1. We perform a first **reduction step**: by the **contextual rule for** λ we can apply β**-contraction** to the inner redex $(\lambda z'.\,(z'\ u))\ \lambda x.\,(t\ z)$ and get the term $\lambda z.\,((z'\ u)\,\{\lambda x.\,(t\ z)\,/z'\})$; this term is α**-equivalent** to $\lambda z.\,((\lambda x.\,(t\ z))\ \ u)$ by definition of substitution.

[2] Available at `http://www.lacl.fr/~polonowski/Develop/DBGen/dbgen.html`

2. We perform a second **reduction step**: by the **contextual rule for** λ we can apply β**-contraction** to the inner redex $(\lambda x.\,(t\ z))\ \ u$ and get the term $\lambda z.\,((t\ z)\,\{u/x\})$; this term is α**-equivalent** to $\lambda z.\,(t\,\{u/x\}\ \ z)$ by definition of substitution.
3. We perform a third **reduction step**: we can apply η**-contraction** and get the term $t\,\{x/u\}$ modulo α**-equivalence**.

Indeed, from the source syntax of ordinary λ-calculus, with the help of Coq `Notations`, we can proceeds to define **val** and **let val** as follows. We start with some notations:

```
Notation "t '[||]'" := (_db_term 0 _xnil t).
Notation "t '[|' x '|]'" := (_db_term 1 (_xcons x 0 _xnil) t).
Notation "'(!' t '!)'" := (named_to_db_term 0 _xnil t).
```

The first notation is the embedding of a de Bruijn term `t` inside a named term with no mapping of the free indices of `t`, this means that any named variable bound over it will be free in `t`. The second one is similar except it maps a variable to the first free index of `t`. The third one is the named to de Bruijn translation function.

As said before, this provides a handy way to deal with variable fresh- and freeness. The first usage we make is to define η-reduction, usually written

$$\lambda x.\,(t\ x) \to t \qquad \text{if } x \text{ is not free in } t$$

Indeed, we can define it in the reduction relation as follows (where `/\X.` is the notation for $\lambda x.$ in the name setting, `@` is for the application and `&X` is for a named variable `X`):

```
Inductive red : term -> term -> Prop :=
...
| eta :
  forall X (t a : term),
    a = (! /\ X.(t[||] @ &X) !) ->
    a |-> t
...
```

This reads almost as in the pencil-and-paper definition, `t[||]` asserting that the variable `X` is not free in `t` (no capture of the indices of `t`). We then define the **val** and **let val** abbreviations as follows:

```
Definition _val (t : term) : _term := /\ "x".(&"x" @ t[||]).

Definition _letval Y (t : term) (u : _term) : _term :=
  /\ "X".(t[||] @ /\ Y.(u @ &"X")).

Notation "'let_val' x '=_' t 'in_' u" := (_letval x t u) (at level 50).
```

Here again, this is a nice improvement in the usage of de Bruijn infrastructure. We can then state our lemma and do the proof easily:

```
Lemma letval_val :
  forall (t u : term),
    (! let_val "y" =_ val u in_ (t[|"y"|]) !) |--> t[u].
Proof.
  intros t u. unfold _letval, val, _val. simpl.
  eapply reds_step. apply ctx_lam. apply beta. simpl; dbgen_tac.
  eapply reds_step. apply ctx_lam. apply beta. simpl; dbgen_tac.
  apply reds_step with (t[u]). apply (eta "X"); simpl; dbgen_tac.
  apply reds_zero.
Qed.
```

Notice that no direct invocation of infrastructure properties is needed, although this proof involves relatively complex lifting and substitution interactions (between those of the abbreviations definition and those created by the reduction steps). All the infrastructure treatment is performed by the `dbgen_tac` tactic, and only the interesting steps remain in the proof script.

This definitely looks close to the pencil-and-paper proof: the proof begins with macros unfolding and simplification, then each `apply ...` or `eapply ...` corresponds to a proof step (performing one reduction step is defined as `reds_step`; the rule for contextual reduction under a λ is `ctx_lam`; reduction rules are `beta` and `eta` respectively), and each `simpl; dbgen_tac` deals with the subgoal of equivalence modulo lifting and substitution definitions. The last `apply reds_zero` concludes the proof with the reflexivity of the multi-step reduction relation noted `|-->` in the lemma statement.

5 Conclusion and Further Work

We have seen that this approach is satisfactory at the user level as it provides an easy way to formalize higher-order languages without having neither to embed it in a complex theory (potentially subject to adequacy problems) nor to encode it as a more complex syntax. It comes with a rather complete generation of the de Bruijn infrastructure, and a set of tactics that makes further proofs involving this infrastructure quite easy. We also believe the proposed translation from the generated named syntax to be a good incentive for users not very familiar with de Bruijn ones. Besides alleviating the burden of formalization, this framework also allows as many modifications of the source language as needed, without having to parse and correct hundred or thousands of lines of formal specifications and proofs.

DBGen is already more than a prototype, it handles complex languages and can be easily extended for further content generation. Forthcoming extension are, at the infrastructure level, the definition of free and bound variables computation functions and the properties of lifting and substitution w.r.t. those notions. At the language level, we plan to add support for indexed relations (like typing relations, or reduction with an environment) and perhaps to automatically provide some commutation lemmas (such as subject reduction). DBGen could also be adapted to generate Locally Nameless syntaxes (as did *LNgen* in a recent past) and translation functions for named and de Bruijn syntaxes.

Acknowledgments. We would like to thank Tristan Crolard for the numerous discussions about this work, and the anonymous referees for their valuable comments.

References

1. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) 75(5), 381–392 (1972)
2. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G.C., Wadler, P. (eds.) Proceeding of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 3–15. ACM (2008)
3. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing 13, 341–363 (2001)
4. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation 186, 165–193 (2003)
5. Urban, C., Tasson, C.: Nominal techniques in isabelle/hol. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 38–53. Springer, Heidelberg (2005)
6. Charguéraud, A.: The locally nameless representation. Journal of Automated Reasoning (2011)
7. Aydemir, B., Weirich, S.: LNgen: Tool support for locally nameless representations (March 2009)
8. Polonowski, E.: DBGen User Manual, TR–LACL–2012–4 This document introduces DBGen, an automatic generator of De Bruijn infrastructure for the Coq proof assistant. TR–LACL–2012–4 (2012)
9. Church, A.: The Calculi of Lambda Conversion. Princeton University Press (1941)
10. Aydemir, B., et al.: Mechanized metatheory for the masses: The poplmark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
11. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press (1990)
12. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. Theoretical Computer Science 121(1-2), 279–308 (1993)
13. Lee, G., Oliveira, B.C.D.S., Cho, S., Yi, K.: Gmeta: A generic formal metatheory framework for first-order representations. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 436–455. Springer, Heidelberg (2012)
14. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strinsa, R.: Ott: Effective tool support for the working semanticist. Journal of Functional Programming 20, 71–122 (2010)
15. Crolard, T., Polonowski, E.: A program logic for higher-order procedural variables and non-local jumps. Technical Report TR-LACL-2011-4, 50 pages (2011-2012)
16. Crolard, T., Polonowski, E.: Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. Journal of Logic and Algebraic Programming, 181–208 (2012)

Shared-Memory Multiprocessing for Interactive Theorem Proving

Makarius Wenzel*

Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

Abstract. We address the multicore problem for interactive theorem proving, notably for Isabelle. The stagnation of CPU clock frequency since 2005 means that hardware manufactures multiply cores to keep up with "Moore's Law", but this imposes the burden of explicit parallelism to application developers. To cope with this trend, Isabelle has started to support parallel theory and proof processing in 2007, and continuously improved the use of multicore hardware in recent years. This is of practical relevance to theory and proof development, since their size and complexity is roughly correlated with the real time required for re-checking. Scaling up the prover on parallel hardware will facilitate maintenance of larger theory libraries, for example.

Our approach to parallel processing in Isabelle is mostly implicit, without user intervention. The system is able to exploit the inherent problem-structure of LCF-style proof checking, although it requires substantial reforms of the prover architecture and its implementation. Thus the user gains significant speedup factors on typical commodity hardware with 2–32 cores; saturation of 8 cores is already routine in many applications.

The present paper provides an overview of the current state of shared-memory multiprocessing in Isabelle2013, which also benefits from recent improvements of parallel memory management in Poly/ML (by David Matthews). We discuss common requirements, problems, and solutions. Concrete performance figures are analyzed for some applications from the Isabelle distribution and the Archive of Formal Proofs (AFP).

1 Introduction

1.1 The Multicore Problem

Software developers have become accustomed to *Moore's Law* of computing, which states that chip density and integrated functionality doubles every two years. This is essentially a social contract of hardware manufactures with its customers, the producers of computer systems and application software. In the past, it was correlated with an increase in clock frequency, so existing programs would become exponentially faster over time, or the complexity of programs could be increased without the user noticing such "software bloat".

* Current research supported by Project Paral-ITP (ANR-11-INSE-001).

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 418–434, 2013.

The rules have changed substantially around 2005, when clock frequency has reached a plateau at 3 GHz — due to excessive power dissipation and overheating at higher rates. Thus the continued exponential growth leads to a multiplication of explicitly visible CPU cores, presently in the range of 2–32.

Multiplication of cores naturally poses challenges to application software development. Sequential code that fails to adapt to this evolutionary pressure suffers from exponential decline of relative performance. For example, a single-core program on 16 cores is confined to 6.25% of the nominal CPU power. Even if the number of cores becomes stagnant, which might well happen especially in the consumer market, we are left with the problem of a large gap in potential application performance, and the era of sequentialism is not coming back.

Multiprocessing does not provide "spare CPU cycles" for free: extra effort is required to use the available CPU power in applications. Performance matters for interactive theorem proving, since the real time spent for re-checking is correlated with the total size of formal developments. Big Isabelle applications (e.g. from the Archive of Formal Proofs) typically grow until the time for full re-checking approaches 10 min to 1 h. It is up to the prover implementation to stretch the amount of formal content that can be processed in that time-span.

An important characteristic of the hardware class with 2–32 cores is that the convenient programming model of *shared memory* can still be supported, with reasonable memory bandwidth for transfers between CPU modules. This requires hardware manufacturers to provide an increasingly complex memory hierarchy of caches and quick paths for physically distributed memory, but it is one area where the exponential increase of hardware capabilities still happens (apart from graphics performance). There is some variance in the different product lines of Intel vs. AMD: in 2013 high-end CPUs by Intel emphasize the performance of shared memory access, while AMD maximizes the number of cores per chip.

For the concrete measurements in this paper, we shall use a 3rd generation Mac Pro (early 2009) with 8 CPU cores and 16 hardware threads (2×4-core hyperthreading Intel Xeon at 2.93 GHz) and 32 GB main memory (DDR3 at 1066 MHz), running Mac OS X Mountain Lion in genuine 64 bit mode. This represents a typical (slightly dated) workstation. The performance of current high-end laptops (e.g. based on Intel Core i7) is only a factor of 2 lower than that — there is usually just one CPU module on mobile systems.

1.2 LCF-Style Provers as Multi-threaded Applications

Due to various characteristics, interactive theorem provers like HOL [19, §1], Coq [19, §4], Isabelle [19, §6], or ACL2 [19, §8] fit quite well into the model of shared-memory multiprocessing.

Functional Programming with Mainly Immutable Data. Typical provers are implemented in a higher-order functional programming language (LISP, ML, Haskell) with a strong emphasis on large symbolic data structures that are immutable (e.g. big λ-terms for syntax or proof terms).

In the multicore era, immutability is one of the inherent advantages of pure functional programming, and even Java programmers have noticed that (cf. the attention that functional-object-oriented Scala and the LISP/Haskell dialect Clojure have gained in the JVM world). Shared memory allows to pass pointers to immutable data without requiring copying, and without the danger of data corruption between application threads.

Moreover, structural equality of pure values (as defined in Standard ML and Haskell) enables the runtime system to produce distributed copies without special precautions about coherence between different processors. This is exploited in the parallel garbage collector of Glasgow Haskell [8] and Poly/ML.

Thus pure functional programs can afford thread-based parallelism, without the hazards known from C or FORTRAN. Nonetheless, threads and synchronization primitives are difficult to use directly in application code, so higher principles of parallel functional programming are required.

LCF-Style Abstraction of Formally Certified Entities. In the LCF architecture [5] that is observed by the HOL family and Isabelle, certified entities like theorems, well-formed terms, and theory certificates are represented as entities of abstract datatypes. The corresponding proof constructions only exist as Platonistic ideas, without representation in memory.

Such abstract datatype values can be easily transferred in shared memory by the runtime system, with the same type-safety properties as the original ML design [4]. In contrast, explicit communication of results between separate process address spaces requires externalization of formal entities. Depending how thoroughly proof checking is treated at the kernel level, this may demand full proof terms to be communicated, say over a network of CPUs.

Continuous Interaction with a Large Prover Process. Our prover interaction model is centered around a single process with a large background context, where the user produces small additions incrementally. This scenario can be efficiently represented by a single multi-threaded process.

Using separate prover processes instead, say via Unix-fork with the usual "copy-on-write" implementation of virtual memory, is faced with some problems. First, the initially shared physical memory map diverges after some run-time of the ML system, notably due to garbage collection that moves equal content in different ways and thus produces separate copies. Second, the results of a fork need to be communicated back by explicit inter-process communication, using some externalized form of proof objects. On non-Unix systems (Windows), startup time of a fresh prover process might be even considered too high in immediate user interaction.

In contrast, the advanced parallel memory management of multi-threaded Poly/ML retains the original structure of data on a shared heap. Recent Poly/ML 5.5 explicitly recovers structural sharing of equivalent data in a long-running ML process, as a special phase in its parallel garbage collection. This allows to scale to more threads working on less memory: 32 bit mode with small 2–3 GB address space has become interesting again for big Isabelle applications, due to reduced memory bandwidth requirements.

These introductory observations should make sufficiently clear that interactive theorem proving and shared-memory multiprocessing are worth investigating and turning into practice. We shall provide a general overview of many questions that arise when embarking on such a project, and provide some clues how the answers of Isabelle could be transferred to other interactive provers.

2 Strategies for Parallel Proof Checking

Subsequently, the running example is the medium-sized entry *Slicing* from AFP (http://afp.sf.net/entries/Slicing.shtml). Its sequential runtime is 12 min.

2.1 Peep-Hole Parallelism

Inspecting the sources of AFP/Slicing reveals the following situation at line 1167 of `Slicing/JinjaVM/JVMCFG_wf.thy`:

```
(* This takes veeery long! *)
by simp_all
```

This solves a goal state of 1225 subgoals by simplification in 77 s elapsed time. Since all subgoals are independent, without any schematic variables whose instantiation could influence each other, it is an "embarrassingly parallel problem". The obvious idea is to simplify these subgoals separately and recombine the results by back-chaining with the original goal state. In Isabelle2013, the proof method *simp_all* is smart enough to detect this situation and to operate in parallel by default. It uses the general-purpose tactical `PARALLEL_GOALS`, based on `Par_List.map` in Isabelle/ML. 1225 simplification tasks are forked, and all results joined before proceeding. Timings for this experiment are given in figure 1. Empirical results of parallel performance need to be treated carefully, looking closely what is measured and how. The figures of elapsed time vs. CPU times are based on standard facilities of the operating system, which we take for granted. The column "pseudo speedup" gives some impression how much nominal CPU cycles are spent, but generally does not tell what the user gains (apart from

worker threads m	elapsed time $\varepsilon(m)$	CPU time $\zeta(m)$	pseudo speedup $\zeta(m)$ / $\varepsilon(m)$	real speedup $\varepsilon(1)$ / $\varepsilon(m)$
1	77.0 s	77.3 s	1.0	1.0
2	38.5 s	76.5 s	2.0	2.0
4	19.7 s	76.8 s	3.9	3.9
6	13.8 s	79.5 s	5.8	5.6
8	10.7 s	80.0 s	7.5	7.2
12	9.1 s	99.1 s	11	8.5
16	8.1 s	113 s	14	9.5

Fig. 1. Simplification of 1225 independent subgoals, with hyperthreading for $m > 8$

extra heat production by the computer). The "real speedup" $\varepsilon(1)$ / $\varepsilon(m)$ represents the success of parallelization more faithfully, but it is usually lower and less exciting in the presentation, and $\varepsilon(1)$ is unknown in practice without the sequential run for comparison. Isabelle batch mode displays the ratio $\zeta(m)$ / $\varepsilon(m)$ as speedup factor by default, and it happens to approximate the real speedup above reasonably well, before the multicore system is pushed towards its limits.

What have we gained so far? Reducing a single tactic application from roughly 80 s to 8 s gives an impressive speedup factor 10, an isolated boost of performance that might be useful for the user working at that spot. Looking at the bigger picture, though, the overall runtime of AFP/Slicing merely shortens from 720 s to 648 s (factor 1.1), as there are no further "embarrassingly parallel problems". This means the total performance improvement with 16 worker threads is not 1000%, but 10%. In other applications it might be as low as 1%.

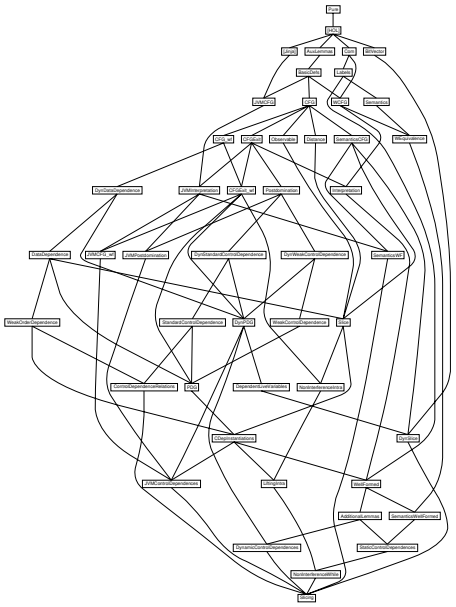
This effect is typical for "peephole parallelization", it applies to most problem domains when the aspect of parallelism is considered naively. *Amdahl's Law* (from 1967) estimates the sub-linear speedup as $1 \;/\; (s + p/m)$, where s is the part of the program running sequentially, and p the part running in parallel (normalized such that $s + p = 1$). For $m \longrightarrow \infty$ this converges to $1 \;/\; s$. In other words, the overall success of parallelization depends on the remaining fraction of inherently sequential code. The prediction would become more pessimistic by including losses due to organization of parallel computation, so for large number of cores the speedup eventually becomes smaller than 1, and ultimately tends towards 0.

2.2 Pervasive Theory and Proof Parallelization

The main conclusion of the previous experiment is that parallelism needs be *pervasive* to gain significant speedup, i.e. the remaining sequential part of the application runtime needs to approach 0. In order to get anywhere close to that we need to investigate our problem structure more thoroughly. We shall do that at different levels of granularity, as specified by parameter q below.

Granularity q = 0: parallel theories.
Typical formalizations consist of an acyclic graph of theory nodes, often with a reasonable degree of independent paths, e.g. see figure 2 (left). Traversing this graph in depth-first order and composing nodes in a bottom-up manner, we gain some potential for parallelism in correlation with the breadth of the graph and the runtime for each node. This resembles `make -j` on the Unix command-line, but we run multiple threads within the same ML process, using our own scheduler for DAG-structured evaluation in Isabelle/ML.

Granularity q = 1: parallel theories and toplevel proofs.
As sketched in figure 2 (right), each theory node consists of a sequence of definition–statement–proof. Results are *specified* beforehand as propositions in the text, and later *justified* by the proofs, which are *irrelevant* in practice. Likewise, some definitional forms require proofs internally (e.g. **inductive**).



inductive P ...

theorem A ⟨*proof*⟩
theorem B ⟨*proof*⟩
theorem C ⟨*proof*⟩

have $X \wedge Y$
proof
  show X **by** *simp*
  show Y **by** *blast*
qed

theorem A_1 **and** ... **and** A_n
  by *simp_all*

Fig. 2. Typical theory dependencies and content structure

Even though proofs are hard to produce and take long to check, they are not required to process the outermost sequence of specifications. Proofs can be forked immediately and are only joined in the very end, when the whole graph of loaded theories is consolidated. Proofs may refer to previous theorems, but not to their proofs.[1]

So the totality of proofs emerging from a given theory graph poses again some "embarrassingly parallel problem" as in §2.1, but with slightly different characteristics: proof problems emerge dynamically during the ongoing theory processing, and need to be joined only in the very end. Instead of static skeletons like `Par_List.map`, the appropriate programming model is that of dynamic *fork* / *join* of eventual results (cf. §3.1).

Granularity q = 2: parallel theories, toplevel proofs, and end-proofs in Isar. Incidentally, the structured proof language Isabelle/Isar [16] provides high degree of compositionality, and thus extra potential for parallel checking. In principle, every Isar sub-proof could be treated recursively like $q = 1$, but for simplicity we only do this for Isar end-proofs "**by** *method*". This is sufficient for structured proof outlines, because most of the time for checking is spent at terminal positions, where claims emerging from top-down decomposition are finally established by arbitrary proof tools (*simp*, *blast*, *auto*, *force* etc.).

Concrete results for these parallelization strategies are given in figure 3. In each column for q, the speedup curve for increasing m flattens according to Amdahl's

[1] Despite the different foundational approach in the Dependent Type Theory of Coq, proof irrelevance still holds in practice, since most proofs are "opaque".

Law. This is inevitable, but it matters how far the relative decline can be postponed. The combination of parallelization strategies for $q = 2$ achieves fairly good speedup of 6.2 on 8 cores, more than 75% of the nominal CPU power.

worker threads m	real speedup $q = 0$	real speedup $q = 1$	real speedup $q = 2$
2	1.6	1.8	2.0
4	2.0	3.0	3.3
6	2.1	3.5	4.1
8	2.2	3.7	6.2

Fig. 3. Real speedup of AFP/Slicing depending on granularity q

The results of this single experiment can be extrapolated — it has been chosen to represent typical Isabelle applications seen today. Thus we conclude the following rules of thumb for multicore scalability:

- Parallel theory loading alone scales to **2 cores**,
- with additional toplevel proof parallelization it scales to **4 cores**,
- with additional sub-structural proof parallelization it scales to **8 cores**.

Another parameter that is not measured here is the level of sophistication of the parallel prover implementation. In 2011 AFP/Slicing did not scale beyond 4 cores, and in 2009 only few Isabelle applications managed to go significantly beyond 2 cores — see [15, §5] for the best that could be achieved on 4-core Intel Xeon hardware in that time.

3 Parallel Prover Architecture

Despite good side-conditions for multi-threaded proof checking (§1.2), substantial reforms of the prover architecture (and its implementation) are required to make it actually work and perform well. This affects a broad spectrum of core prover aspects: parallel functional programming, parallel inference kernel, explicit organization of theory and proof structure.

There is further impact on outer system integration layers. For example, Isabelle2013 provides an advanced build system (implemented in Isabelle/Scala) to manage re-checking of large theory libraries efficiently, by managing a tree of multi-threaded processes that run in parallel. Thus by exploiting the outer hierarchy of "sessions" and the inner structure of theories and proofs, full re-checking of AFP has been reduced from several hours to 30 min on 8 cores.

Further integration of parallel checking and asynchronous interaction happens in the Prover IDE [17]. In Isabelle2013 it exploits more fine-grained proof parallelism during regular interaction. Combining erratic edits by the user and continuous parallel checking by the prover poses further challenges that are beyond the scope of the present paper (some aspects are discussed in [18]).

Subsequently, we provide an overview of Isabelle2013 prover architecture, putting it into perspective of earlier work and pointing out recent refinements.

3.1 Parallel ML

LCF-style theorem proving has been intertwined with functional programming in ML since the inception of both in LCF [4]. Originally implemented as an interpreted language within LISP, ML has become a standalone language in the 1990-ies. Some of its implementations have managed to catch up with multithreading already, such as Poly/ML [9] (with its own parallel runtime system) and F# (which re-uses the common language runtime of the .NET platform). Other interesting functional languages with good support for parallel programming are Haskell [8] and Scala, and of course LISP where important ideas of task-parallel evaluation was first explored in the 1980-ies [6].

The most notable exception is OCaml, which is still subject to early decisions by its main architects of *not* supporting parallel threads in ML, despite [3].

The host language for particular provers is inherited from distant past and somehow accidental, but it impacts chances of survival in the multicore era. Isabelle has always supported more than one implementation of Standard ML, especially Poly/ML and SML/NJ. Starting in 2006, David Matthews made substantial renovations for Poly/ML to support native multithreading. The Poly/ML 5.5 release from September 2012 is notable for its support of parallel garbage collection and compaction of large heaps. There is now a considerable performance gap towards SML/NJ: in 2013 the factor is of the order 10^2 for medium-sized Isabelle applications, and big ones are already infeasible.

In a system like Poly/ML, the raw power of shared-memory multicore hardware is made available as *threads–and–locks*. Poly/ML offers an ML view on POSIX threads, with its mutexes and condition variables for synchronization and signaling [9, §2]. This first approximation to parallel computation is then augmented by a concept for task parallel programming which organizes evaluation of *future values* in Isabelle/ML [9, §3]. Threads do not scale beyond 10^1–10^2, but a limited number of worker threads can operate efficiently on a task queue of 10^5–10^6 pending evaluations. The idea of data-oriented parallelism dates back to Multiplisp [6] at the least. It has been re-implemented over decades in many variations, and is routinely available in Isabelle/ML, F#, Haskell, Scala, LISP.

The Isabelle/ML implementation of futures is careful to transfer the semantics of Standard ML adequately into the parallel environment, with strict functional evaluation, synchronous program exceptions, and asynchronous interrupts. The main programming interface is as follows:

type α *future*
val *Future.fork*: $(unit \to \alpha) \to \alpha$ *future*
val *Future.join*: α *future* $\to \alpha$
val *Future.cancel*: α *future* $\to$ *unit*

Type α *future* represents the eventual result of a given expression, which is associated with an evaluation task of the future scheduler in the background. The task queue supports both priorities and dependencies, and is implemented as explicit graph structure. Joining with an unfinished future synchronizes with the evaluation process, where the full complexity of inter-thread communication happens. In contrast, there is no special overhead to access finished futures later.

The key property is that *Future.join* (*Future.fork* (**fn** () ⇒ *expr*)) produces the same result (or exception) as *expr* outright, but the fork can be separated from the join for parallel evaluation. The overhead for fork + join is about 10^{-5} *s*. On Intel hardware similar to our's, Rager [13] reports 50 μs for his LISP system, and we measure exactly the same for Isabelle/ML. This overhead roughly determines the granularity of tasks that are feasible to fork. For example, 20000 tasks that run a few microseconds each will waste 1 s, but this not a problem if the application manages to produce thousands of tasks in the range of milliseconds.

Futures can be used to implement higher-level combinators like parallel `map` (or the renowned `reduce`). Isabelle/ML already provides such derived combinators, but proof parallelization uses *Future.fork* and *Future.join* directly, because proofs emerge dynamically during the exploration of yet unknown theory content. Since forks are under program control, we can exploit potential parallelism while exploring the proof, and easily restrain substructural parallelization tasks.

3.2 Theory Context and Proof Promises

Logical derivations work in a context, whose precise structure depends on the formulation of the underlying logic. In the HOL family and in Isabelle, there is a *global theory context* that we call Θ, and a *local proof context* that we call Γ.

The theory Θ contains declarations and specifications of type constructors, term constants, and axioms (definitions), which are polymorphic in the sense that their type schemes may get instantiated arbitrarily during proof.

The proof context Γ contains local hypotheses (premises) to support $\Longrightarrow$ introduction in Natural Deduction. Locally fixed parameters for $\bigwedge$ quantifier introduction are implicit (in contrast to dependent type theory in Coq).

The inference kernel produces sequents $\Theta, \Gamma \vdash \varphi$, but we have $\Gamma = \emptyset$ for global results and the background theory Θ is managed implicitly. Thus end-users may think just of theorems $\vdash \varphi$ that establish a certain proposition φ.

Nonetheless, the global context Θ turns out as essential for management of forked proofs in parallel Isabelle. Lets say that at stage Θ_1 of the ongoing theory development, theorem φ is claimed and its proof forked for independent checking, while the theory is continued monotonically towards Θ_2, adding more definitions and theorems. When the forked proof is eventually joined, it needs to establish $\Theta_1 \vdash \varphi$ in the original theory context for proper foundation of logical results.

This means the inference kernel needs to work explicitly with theory contexts, with some operations to extend, merge, compare theories according to $\Theta_1 \subseteq \Theta_2$, and transfer of theorems from the smaller to the bigger theory.

Incidently, Paulson [11] had already introduced a notion of theory context for theorems in Isabelle89. This was motivated by the *logical framework* approach of that time, to allow the user to work in different background contexts. The concept has been refined many times, notably for efficient checking of $\Theta_1 \subseteq \Theta_2$ via symbolic theory certificates that represent the stages of extend and merge operations, without inspecting the theory content directly.

Another important aspect is extra-logical *theory data*: concrete syntax, hints for proof tools etc. are managed in a value-oriented manner as part of Θ (and Γ) in the implementation. Thus tools may run in parallel and refer safely to their private data within the context, without worrying about mutable state.

This is in contrast to original LCF and members of the HOL family, who accumulate theory content as implicit global state of the ML process. The state grows monotonically as the user adds new definitions, without returning to earlier states (undo) and without isolation of independent instances of proof tools. Coq is slightly more flexible by multiplexing different contexts, with operations *freeze* and *unfreeze* for tool hints that are associated with them, although this concept is still restricted to a single active view of the state.

These observations reveal accidental side-conditions in the long history of LCF-style provers. In the past, the interaction model was that of a single-threaded TTY loop where individual commands were applied one after another, so simplistic treatment of the context could be afforded. But this should not prevent Coq and HOL systems to become more stateless and timeless eventually.

Taking sequents $\Theta, \Gamma \vdash \varphi$ with explicit context values for granted in Isabelle, we can proceed in the next stage to support a notion of *proof promises* natively in the inference kernel. The new rule *promise* produces a hole (with specified result) in the reasoning, which can be amended later by another rule *fulfill*. Holes are managed formally by the context Π that maps identifiers of proof promises to actual derivations. The original version from 2008 of this slightly extended Natural Deduction system for Isabelle/Pure is given in [15, §3]; according to Isabelle2013 the main rules are as follows:

$$\frac{\mathrm{FV}\, A = \emptyset \quad \mathrm{TV}\, A = \{?\overline{\alpha}\}}{\Theta, \{a : A\}, \emptyset \vdash a[?\overline{\alpha}] : A[?\overline{\alpha}]}\ (\textit{promise})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p : B \quad \Theta_0, \emptyset, \emptyset \vdash q : A \quad \Theta_0 \subseteq \Theta}{\Theta, \Pi - \{a : A\}, \Gamma \vdash p[a := q] : B}\ (\textit{fulfill})$$

The underlying formulation of Isabelle/Pure with proof terms goes back to [2]; it emphasizes the role of proof promises as polymorphic proof constants, which may be substituted by closed proof terms later. In reality, proof terms are merely a second option of the Isabelle kernel. By default it only maintains a *proof body* as a digest of proof promises, oracles, and external theorem references.

A notable refinement of *fulfill* compared to [15, §3] is that the replacement proof $q : A$ is required to be fully closed ($\Pi = \emptyset$). This avoids complications of holes depending on other holes, and speculative well-founded ordering of the same, which would be hard to implement in practice. The restricted form means that future proofs need to be joined in a bottom-up manner before passing through the inference kernel. The well-founded order is given by the physical process of Isabelle/ML consolidating values. In best LCF tradition, this might lead to non-termination, but cannot produce unfounded results.

Management of free type variables is explained further in [15, §3]. It is possible to quantify term variables and thus demand $\mathrm{FV}\ A = \emptyset$ w.l.o.g., but type variables need to be tracked separately to retain the schematic polymorphism of

the logic. Otherwise it would be impossible to fork a proof and instantiate types of the result. Our context Π acts like Θ in this respect. This is in contrast to shallow proof holes of [1], which are monomorphic due to the use of Γ in HOL.

The ML interface of the inference kernel presents *promise* and *fulfill* together as a single operation *Thm.future*: *thm future* $\rightarrow$ *term* $\rightarrow$ *thm*. It makes a theorem based on a proof promise that will be fulfilled by the eventual result of the given future. The side-conditions are checked on finished future derivations. Thus the implicit policy of future evaluation is re-used, but the kernel stays in control of checking the outcome as plain ML value; it does not even care about parallelism.

ML values of type `thm` are theorems on the surface only, as they may depend on unfinished futures. The kernel operation *Thm.join_proofs*: *thm list* $\rightarrow$ *unit* consolidates results by recursive joining of the graph of pending proof promises, and produces a digest of parallel error messages about failed proofs.

3.3 Goals with Forked Proofs

The notions of proof promises and future theorems of the inference kernel mainly serve foundational purposes in the spirit of the LCF architecture. The main programming interface works via goals with forked proofs. There is some additional infrastructure for accounting and reporting of structural errors within forked proofs. As already observed in [15, §4], the Isabelle/ML operation for goal-directed proof *Goal.prove*: *Proof.context* $\rightarrow$ *term* $\rightarrow$ *tactic* $\rightarrow$ *thm* can be turned into an alternative version *Goal.prove_future* of the same signature. Its internal use of future theorems is hidden, thanks to the full specification of the intended result as proposition. This is a key advantage of backward-proof.

There are delicate differences in the semantics of proof failure, though, if errors in forked proofs are postponed until the final join over all theories. This is important for derived definitional packages like **inductive** in Isabelle/HOL, where failure of its internal monotonicity proof means that the user specification is malformed. Any further derivations inside **inductive** are irrelevant to the user: they always work under the assumption that the package is implemented properly. This critical treatment of forked proofs was still relatively crude in 2008 [15], resulting in more conservative use of sequential proofs in some situations.

To avoid such conflicts of parallelism and reliability of the prover, the high-level infrastructure for goal-directed parallel proof has been reworked significantly for Isabelle2013. The main aspects are summarized as follows:

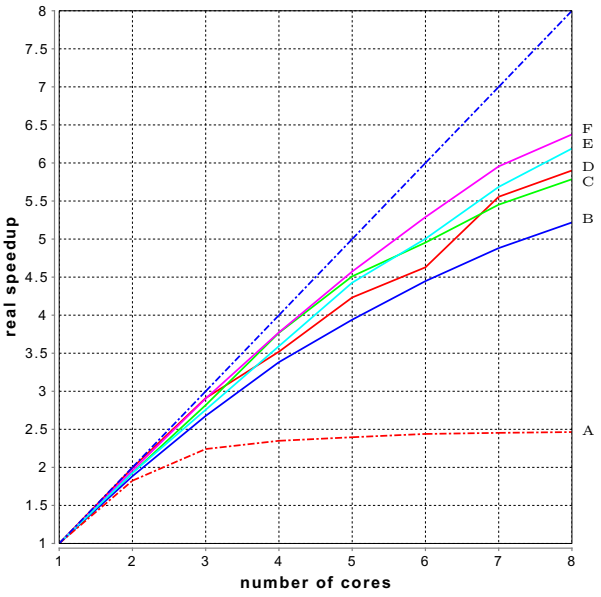
- λ-lifting wrt. to the proof context Γ according to [15, §4.1], to allow goal statements work with premises and parameters, despite the restrictions of *Thm.future_thm* due to the *promise* and *fulfill* rules (§3.2).
- Global accounting of forked proofs within the running process, to avoid unnecessary forks when the system is flooded with future tasks already, according to the bound of $m *$ *parallel_proofs_threshold* (default 100).
- Systematic tracking of errors stemming from forked proofs according to the originating command transaction.

Consequently, derived elements like **inductive** and **datatype** may now fork their internal proofs more aggressively, even relevant ones. This is especially important for increased parallelism in the asynchronous document model of the Prover IDE. An explicit notion of *stable command* in its document model indicates the status of all goal forks, without requiring a global join. The system will reset failed command transactions whose forked proofs were failing or interrupted, and thus retry evaluation in the next editing phase.

4 Performance and Scalability

Asymptotically, the multicore problem cannot be solved, but we do our best to exploit the capabilities of our hardware. Subsequently we review further results of measuring the parallel performance, to see trends beyond 8/16 cores. Current Poly/ML 5.5 and Isabelle2013 allow monitoring of CPU and memory usage, status of parallel garbage collection, future tasks and worker threads.

For users the main result is the real speedup $\varepsilon(1)$ / $\varepsilon(m)$, which is presented in figure 4 for various Isabelle sessions.



		$\varepsilon(1)$	$\varepsilon(8)$
A	*HOL*	210 s	85 s
B	*HOL-UNITY*	69 s	13 s
C	*HOL-Nominal_Examples*	526 s	91 s
D	*Slicing*	720 s	122 s
E	*HOL-Decision_Procs*	415 s	67 s
F	*HOL-Hoare_Parallel*	218 s	34 s

Fig. 4. Sequential runtime and real speedup of some Isabelle sessions

Session HOL is special here in compiling big ML modules for tools like Sledgehammer, and comparatively few regular theory and proof developments. Factor 2.5 for 8 cores might look disappointing, but it is already an improvement over 1.7 in [15, §5], where the base-line performance was much lower as well.

The other sessions are more conventional, with good speedup in the range of 5.2–6.4 for 8 cores. To scale further, potential losses in the implementation of the parallel ML infrastructure are only of minor concern: they can be ironed out eventually. The main challenge is proper partitioning of parallel tasks according to the structure of the application. The histograms in figure 5 illustrate distribution of the runtime of tasks for $m = 8$ and $q = 2$ (cf. §2.2).

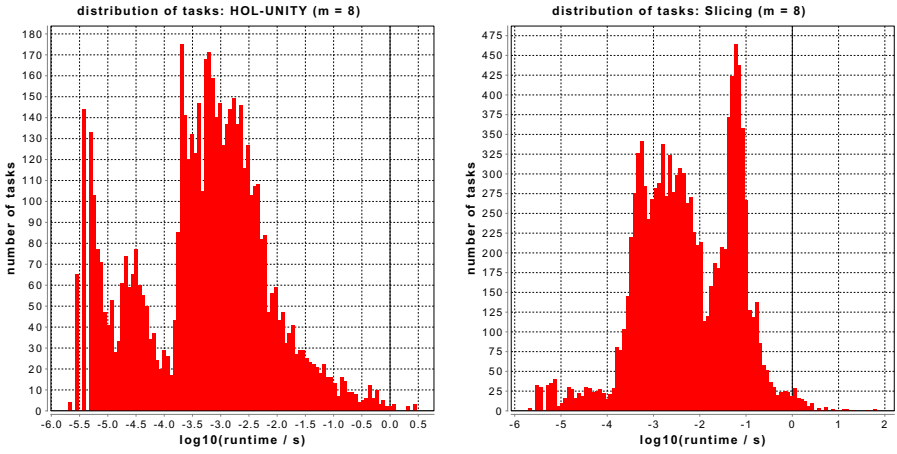


Fig. 5. Distribution of task runtimes ($m = 8$)

The examples HOL-UNITY vs. AFP/Slicing are very different in their absolute runtime and overall structure of theories and proofs. Both scale to 8 cores, which can be explained by a good amount of tasks in the millisecond range. AFP/Slicing tends to more longer-running proofs (many slow automated steps), with a few monolithic tasks in the range 10^1–10^2 s.

In figure 6 we see more precisely how this portfolio of future tasks populates the Isabelle/ML task queue and corresponding worker threads, over the elapsed runtime of each session. This changes significantly for different values of m.

The fluctuation of thousands of ready tasks is mainly due to forked proofs. These easily saturate 4 worker threads during most of the elapsed runtime, but for 8 there are increasing drop-outs with inactive workers: HOL-UNITY has a slow startup-ramp, until sufficiently many proof tasks are forked; AFP/Slicing has a slow tail-end with a few long-running tasks. The final theory of AFP/Slicing consists of one huge proof, with many automated steps; this is where many of its 10^1–10^2 s tasks emerge. The sudden peak near the start of AFP/Slicing is due parallel checking of 1225 subgoals, as discussed in §2.1.

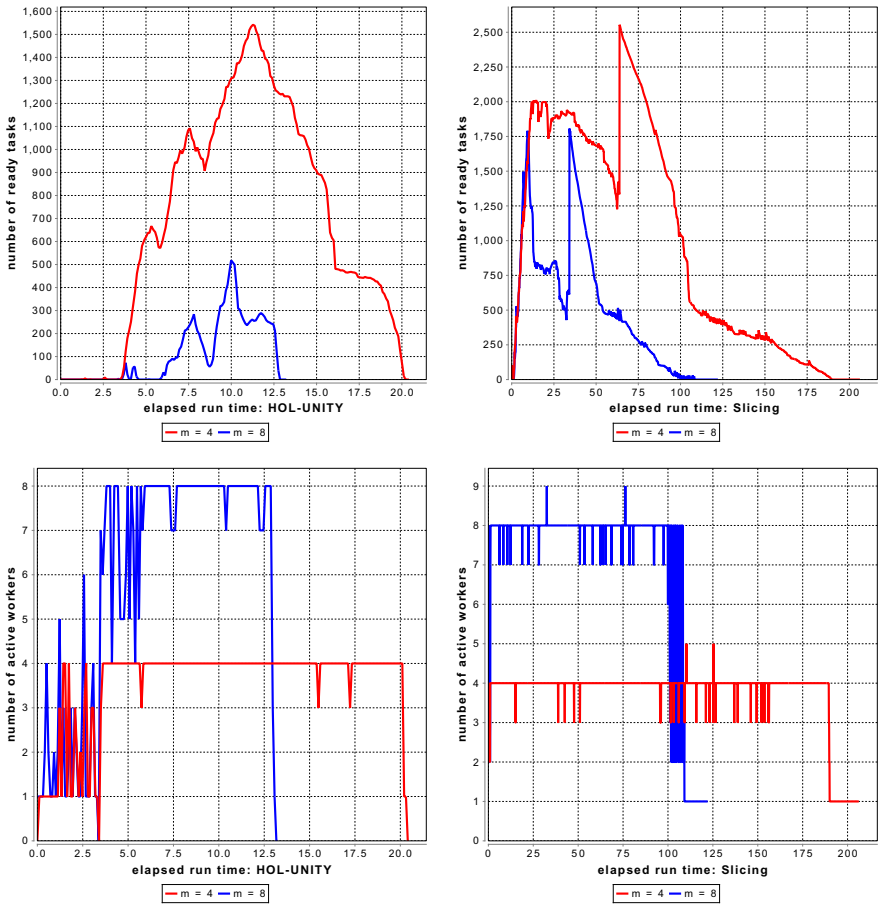


Fig. 6. Task queue population and worker thread utilization ($m = 4$ and $m = 8$)

5 Conclusion

We have demonstrated that proof checking in the LCF tradition can exploit the computing power of shared-memory multicore hardware adequately. Our main strategy for parallelization is based on the observation that formal theories consist of explicit statements with irrelevant proofs. Additional aspects of Isabelle/Isar sub-proofs are important for further scalability.

Implementations of parallel ML should in principle be commonplace, but we had to rebuild significant infrastructure for SML from scratch (starting in 2006/2007), based on parallel Poly/ML provided by David Matthews. Versions of HOL that are also implemented in SML could re-use that, but some prover-specific infrastructure needs to be reworked. Coq faces more serious challenges

since its home platform OCaml is optimized for sequential execution, although its predecessor Caml once had a multithreaded runtime system [3].

To relate performance figures of earlier versions of parallel Isabelle, note that [15, §5] also uses the top-end Mac Pro at that time (4-core Intel Xeon), but [9, §5] is different in using a fat-node of a computing-cluster (32 AMD Opteron CPUs and 64 GB main memory); the measurements of [9, §5.3] stretch the available resources, to explore the limits of ML memory management and application task structure [9, §5.3]. The present results require much less memory, and work even within the restricted address space of the 32-bit version of Poly/ML 5.5.

The challenge of explicit parallelism in application code has happened before in the late 1980-ies and early 1990-ies, when classic CISC machines became stagnant and "transputers" or workstation clusters were considered a viable alternative to gain more performance. Some parallel prover projects from that time include the *Distributed Larch Prover* [7], or the *MP refiner* [10] as parallel tactical engine for NuPrl, using an extinct parallel version of SML/NJ. This pioneering work had little impact on mainstream interactive provers later on, and the boost of performance of RISC machines and reformed CISC machines has postponed the problem of mainstream parallelism until 2005.

The only other major proof assistant that answers the multicore challenge is ACL2: Rager [12, 13, 14] provides parallel execution within the LISP system, and reworks the main stages of the interactive proof development process ("the ACL2 waterfall") to support parallelism in many practical situations. Performance is evaluated into the range of 32 cores on latest Intel hardware. ACL2 6.0 from December 2012 includes the parallel variant ACL2(p) already.

ACL2(p) emphasizes parallel enhancement of interactive proof discovery and case-splitting. This roughly corresponds to our sub-structural Isar proof parallelization, as far as it is already supported in Isabelle2013, but the side-conditions of the proof languages are quite different. Isabelle/Isar emphasizes fast rechecking of structured proof texts, while ACL2(p) emphasizes the search involved in its "waterfall" of interactive proof exploration.

As cores continue to multiply at an exponential rate, our prover infrastructure needs to catch up by more sophisticated parallelization strategies: the multicore problem poses a new challenge for every power of 2 in the CPU multiplication phenomenon. The advanced monitoring facilities of Isabelle2013 will help to isolate bottle-necks in the granularity of future tasks.

We anticipate further sub-structural proof parallelization, exploiting virtues of the Isar proof language more thoroughly: recursive proof forking to accommodate nested Isar proofs that spend substantial time in their outline structure. Another possibility is to introduce more explicit parallelism in specific proof tools, including parallel proof search. Isabelle/ML already provides combinators like `PARALLEL_CHOICE` for tactics, or `Par_List.exists` for generic ML functions.

As the Isabelle Prover IDE manages to support more and more parallelism in its asynchronous editing process, we expect significant shifts of paradigms how large proofs are developed, beyond the raw speedup from the underlying parallel hardware. This requires users to get acquainted with a timeless and stateless

model of document-oriented proof development, and to give up manual control in "driving" the prover in single sequential steps. An advanced proof assistant acts like system software in this respect, and does the parallel scheduling implicitly and automatically without user intervention.

References

[1] Amjad, H.: Shallow lazy proofs. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 35–49. Springer, Heidelberg (2005)

[2] Berghofer, S.: Program extraction in simply-typed Higher Order Logic. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 21–38. Springer, Heidelberg (2003)

[3] Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML. In: 20th ACM Symposium on Principles of Programming Languages (POPL). ACM press (1993)

[4] Gordon, M., Milner, R., Morris, L., Newey, M.C., Wadsworth, C.P.: A metalanguage for interactive proof in LCF. In: Principles of programming languages, POPL (1978)

[5] Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)

[6] Halstead, R.H.: Multilisp: A language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. 7(4) (1985)

[7] Kapur, D., Vandevoorde, M.T.: DLP: A paradigm for parallel interactive theorem proving (1996)

[8] Marlow, S., Peyton Jones, S.L., Singh, S.: Runtime support for multicore Haskell. In: Hutton, G., Tolmach, A.P. (eds.) 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009). ACM (2009)

[9] Matthews, D., Wenzel, M.: Efficient parallel programming in Poly/ML and Isabelle/ML. In: ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP 2010 (2010)

[10] Moten, R.: Exploiting parallelism in interactive theorem provers. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 315–330. Springer, Heidelberg (1998)

[11] Paulson, L.C.: Isabelle: the next 700 theorem provers. In: Odifreddi, P. (ed.) Logic and Computer Science. Academic Press (1990)

[12] Rager, D.L.: Adding parallelism capabilities in ACL2. In: Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2006). ACM (2006)

[13] Rager, D.L.: Parallelizing an Interactive Theorem Prover: Functional Programming and Proofs with ACL2. Ph.d. dissertation, University of Texas at Austin (2012), http://www.cs.utexas.edu/~ragerdl/papers/dissertation/dissertation.pdf

[14] Rager, D.L., Hunt Jr., W.A., Kaufmann, M.: A parallelized theorem prover for a logic with parallel execution. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 435–450. Springer, Heidelberg (2013)

[15] Wenzel, M.: Parallel proof checking in Isabelle/Isar. In: Dos Reis, G., Théry, L. (eds.) ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009). ACM Digital Library (2009)

[16] Wenzel, M.: Isabelle/Isar — A generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof — Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar, and Rhetoric, vol. 10(23), University of Białystok(2007)

[17] Wenzel, M.: Isabelle/jEdit — A Prover IDE within the PIDE framework. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS, vol. 7362, pp. 468–471. Springer, Heidelberg (2012)
[18] Wenzel, M.: READ-EVAL-PRINT in parallel and asynchronous proof-checking. In: User Interfaces for Theorem Provers (UITP 2012). EPTCS (2013)
[19] Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)

A Parallelized Theorem Prover for a Logic with Parallel Execution

David L. Rager, Warren A. Hunt, Jr., and Matt Kaufmann

The University of Texas at Austin, Department of Computer Science
2317 Speedway, Stop D9500, Austin, TX 78712
{hunt,kaufmann,ragerdl}@cs.utexas.edu

Abstract. In order to take best advantage of modern multi-core systems, interactive theorem provers need to parallelize execution effectively. We describe our modification to a particular theorem prover, ACL2, to use parallel execution automatically in its proof process. Since the ACL2 prover is written primarily in the ACL2 programming language, our approach to parallelization takes advantage of ACL2 language primitives for parallel execution. We demonstrate that the resulting system often provides earlier useful feedback from failed proofs and significant reduction of execution time for successful proofs. Thus, our system not only incorporates parallelism into its proof process, but it also provides a platform for writing and verifying parallel programs written in the ACL2 programming language.

Keywords: parallel theorem proving, parallel execution for a formal logic, functional language, ACL2.

1 Introduction

The ACL2 theorem-proving system [1] is used in large industrial formal verification efforts [2,3,4] that continue to drive its development. ACL2 is written primarily in its own functional language, based on an applicative subset of Common Lisp. Our contribution is ACL2(p): by extending the ACL2 programming language with parallelism primitives [5,6,7] and modifying its associated prover, we have enhanced ACL2 to take advantage of multi-core machines and perform simultaneous proofs on dynamically-created subgoals. Our parallelization not only reduces the duration of proofs, but it also provides early feedback for failed proof attempts, which, in turn, can reduce user time required to formulate new lemmas to guide subsequent proof attempts. The addition of parallelism primitives to the ACL2 language enables not only parallelism during proofs, but also proofs *about* parallel programs written in that language. Our focus for ACL2(p) has been on the CCL implementation of Common Lisp, but we have preliminary implementations for two others that also provide native threads: SBCL and LispWorks.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 435–450, 2013.

We make use of the ACL2 regression suite [8,9] to determine the utility of theorem prover parallelism at the subgoal level. Our key results include performance statistics and analysis for these proofs on a contemporary 32-core Intel-based E5-4650 machine. These include proofs about a wide variety of systems and protocols, including one for a theorem about the Java Virtual Machine [10] that experiences a speedup of 25.78x and one for a theorem about deadlock detection [11] that obtains a speedup of 6.21x.

The average factor by which the 200 longest-running proofs are sped up is 5.13x. Reducing the total duration of proof attempts is one way that parallel execution provides earlier feedback to users. A second type of early feedback occurs from concurrently executing the proofs of two or more subgoals. Suppose we have a proof with two independent subgoals, where the first subgoal's proof takes ten seconds and succeeds, and the second subgoal's proof attempt takes one second and fails. In a serial execution starting with the first subgoal, the user has to wait eleven seconds to obtain feedback from failure. To improve this situation, ACL2(p) starts both subgoals immediately, so with two or more CPU cores feedback is provided in only one second.

This paper begins with a discussion of other uses of parallelism in theorem provers. After introducing ACL2's parallel programming primitives, we discuss our parallelization of ACL2's main proof process. We then analyze different types of proofs with respect to parallelization and present our experimental results. Finally, we discuss future work and conclude. Further details of closely-related topics are available elsewhere [12], including the parallelism library's implementation, interactive issues, related work on parallelism for programming languages, performance results on a variety of smaller machines, how we limit parallelism, our methodology used during system development, and how hyper-threading and garbage collection affect performance.

2 Related Work

Several theorem proving systems have supported the parallel execution of different parts of a proof attempt. Some examples are Moten's parallel interactive theorem prover MP refiner [13], Maude's implementation of concurrent rewriting logic [14], the Peers distributed theorem proving prototype [15], the Distributed Larch Prover [16], Partheo [17], and SiCoTHEO [18]. Closest to our work is Wenzel's parallelization of Isabelle/Isar [19,20,21], but first we review parallelism in ACL2 and its predecessor, NQTHM [22].

ACL2 has long taken advantage of the process-level parallelism provided by GNU Make to execute proofs in parallel for a set of files. Of course, this file-level granularity does not speed up an interactive attempt to prove a single theorem. A 1989 report [23] discusses an NQTHM utility for dispatching prover calls in parallel for a set of theorems, but this theorem-level parallelism also did not speed up proof attempts for individual theorems.

A Boyer-Moore style rewriter was the target of an application of Multilisp [24], Qlisp [25], and Parcel [26,27], which compared the ability of the Parcel compiler

to automate the discovery of opportunities for parallel execution with the manual use of `future` and `qlet`. We considered parallelizing execution within the ACL2 rewriter, but our preliminary experiments in that direction were discouraging, so our focus is on parallelism at the subgoal level.

Wenzel's parallelization of Isabelle/Isar [19,20] includes three main opportunities for parallel execution in that system ([19], Section 5.1). The first two are similar to ACL2's file-level and theorem-level parallelism, described earlier in this section. The third opportunity involves checking subcomponents of a theorem's proof. Wenzel calls this parallelism at the *sub-proof* level [20] — this is analogous to our subgoal-level parallelism — and states that he has parallelized the checking of Isar proofs but not the search for Isar proofs ([19], Section 1.3). The point seems to be that individual subgoals presented explicitly by the user are parallelized, but not the proof search performed *within* the proof of a specified subgoal. By contrast, ACL2 (and ACL2(p)) heuristics generate subgoals *dynamically*, beyond what is submitted directly by the user; and for any goal that appears, ACL2(p) generally proves its generated subgoals in parallel (when parallelism is enabled). In summary, our work contrasts with the Isabelle/Isar parallelization work in that we parallelize subgoals that are generated during the search for a proof. A second difference is the capability provided by ACL2(p) for reasoning about parallel programming primitives. By contrast, the futures library in Isabelle is only available in the programming language used to implement the prover — it is not available for programming in the object logic, and one can not reason about what it means to have a future. Finally, our work may be more amenable to scaling: we report a speedup in excess of 25x on a 32-core machine for some proofs, which contrasts with speedups not exceeding approximately 6.5x on 16 cores reported for Isabelle [19].

We are unaware of any use of parallelism in other interactive proof assistants currently being used, including Coq [28], HOL4 [29], and PVS [30,31].

3 Parallel Programming Primitives

ACL2(p) introduces programming primitives that are embedded in the ACL2 logic and enable ACL2 programs to execute in parallel. We designed these primitives for both logical transparency and efficient parallel execution, so that ACL2 programmers can obtain the benefits of parallel execution without complicating their proofs with implementation details like threads and signaling mechanisms. Below we describe one primitive, `plet`, which is a variant of `let` that permits parallel execution, but with no change in reasoning since it is semantically equivalent to `let`. Other parallelism primitives include boolean operators with early termination (`pand`, `por`), parallel argument evaluation (`pargs`), and a combination of such capabilities (`spec-mv-let`) [5,6,7,32,12].

We illustrate the simplicity of `plet` with the following two definitions of the Fibonacci function, one that uses `let` for the recursive calls and the other that uses `plet`. Note that the one that uses `plet` includes a *granularity form*, which is used to restrict parallel execution to sufficiently large inputs. As expected, ACL2

proves automatically that `(fib x)` is `equal` to `(pfib x)` and reports that the proof takes "0.01 seconds."

```
(defun fib (x)
  (cond ((or (zp x) (<= x 0))
         0)
        ((= x 1) 1)
        (t (let ((a (fib (- x 1)))
                 (b (fib (- x 2))))
             (+ a b)))))
```

```
(defun pfib (x)
  (cond ((or (zp x) (<= x 0))
         0)
        ((= x 1) 1)
        (t (plet (declare (granularity
                           (> x 30)))
                 ((a (pfib (- x 1)))
                  (b (pfib (- x 2))))
                 (+ a b)))))
```

We use threads to implement our parallelism primitives, as these incur less overhead than processes. We also "recycle" threads: instead of spawning a new thread for each subgoal, the system places subgoals in a *work queue* and ensures that there are enough threads in existence ready to execute the proofs of these subgoals; after a thread finishes a subgoal, the thread waits for another subgoal to process, and if none arrives, it terminates and becomes available to be garbage collected.

4 Parallelizing ACL2's Main Proof Process

Preparing ACL2's main proof process for parallel execution was a significant portion of this project. Knowledge of some of our difficulties and solutions may help others who wish to parallelize the execution of an interactive theorem prover. We begin our discussion with an introduction to ACL2's main proof process and where we incorporate parallel execution.

The main ACL2 proof process is implemented by the application of nontrivial, heuristic *prover steps*. Each prover step takes a clause and attempts to produce zero or more clauses with the property that if each produced clause is a theorem, then so is the input clause. If the attempt fails, then the clause is saved to a *pool* of clauses whose proofs will later be attempted with induction. But if the attempt succeeds, a prover step is (recursively) applied to each produced clause. When there are at least two produced clauses and there are parallelism resources available, ACL2(p) applies these prover steps in parallel (using `spec-mv-let`), providing *subgoal-level parallelism*. When no clauses remain to be processed, the proof is complete if the pool is empty, and otherwise a proof by induction is attempted with the clauses remaining in the pool. Each induction proof again employs the main ACL2 proof process, typically resulting in more parallelism.

A key problem in parallelizing a system is discovering an appropriate level of granularity for parallelization. As previously described, parallel proof had already been used in ACL2 at the level of files; but we are concerned here with direct interaction with ACL2, in its read-eval-print loop. Furthermore, preliminary experiments indicated that parallelizing the rewriter did not show much promise. Finally, we determined that parallelizing proofs at the subgoal level would incur insignificant overhead, with only 0.58% of subgoals observed to take

less time than the overhead associated with parallelizing a computation ([12], Section 8.3.1). Thus, we decided to parallelize execution at the subgoal level.

Despite the relatively insignificant overhead, when a user stressed an early version of ACL2(p) by producing tens of thousands of subgoals, the proof attempt sometimes caused the machine to reboot ([12], Section 8.3.2.3). Unlike other proofs with many subgoals, in this case the task load average (as reported, for example, by the Linux "top" utility) for the machine increased to the point that the Linux daemon "Watchdog" decided that the machine must have become unstable and rebooted the machine. To solve this problem we changed four aspects of our implementation. First, we placed a limit on the total number of subgoals that can be enqueued for parallel execution, that can already be executing, or that are waiting on other subgoals to finish executing. Beyond this limit, the subgoals prove serially; but this is not a hindrance in practice, as this limit is reached only in extreme circumstances. Second, we changed the default parallelism mode to limit the length of the work queue to three times the number of CPU cores in the system. Third, we changed the implementation of the main proof process from a list-based linear approach to a hierarchical approach that allows the parallelism system to reclaim more quickly the underlying parallelism resources (threads). Finally, we changed how long a thread waits for a subgoal to process before terminating itself, from a constant 15 seconds to a random number between 10 and 120 seconds. This last optimization addresses an interaction between the underlying Lisp and the OS — it prevents thousands of threads from waking up simultaneously, which can cause a spike in the load average for the machine and result in the previously mentioned reboot.

Another difficulty that any developer of a parallel system will face is ensuring that code is free from race conditions. Although the ACL2 theorem prover is written in a language with functional semantics, there are constructs involving a single-threaded *state*, such as output, that produce side effects. Our work involved modifying the code that implements ACL2's main proof process to be thread-safe. As an example, we removed side effects from the mechanism that translates a user-level term into its internal representation (2138 lines in the resulting ACL2 6.0 source code for the primary routines).

When parallelizing a system, a key behavior to preserve is the output provided. ACL2 has long provided an English proof narrative. But such a narrative could look awkward when parallel execution changes the order in which subgoals are processed, and reconstruction of the original order could delay useful feedback to the user about how to proceed from failure. Fortunately, the default mode of ACL2 only prints *key checkpoints*: clauses where the theorem prover becomes "stuck" and where intervention from the user would be most helpful (see ACL2 documentation [5] topics "the-method" and "introduction-to-key-checkpoints"). Furthermore, the order in which key checkpoints are presented is not important. Thus, we have implemented a similar capability for ACL2(p). We use locks for proof output to avoid nonsensical interleaving of characters. But since relatively little printing typically occurs, even for large proofs, locking does not significantly impact performance for parallel execution.

5 Proof Parallelism Potential

The four informal categories below describe proofs according to how they lend themselves to more or less parallelism. They form a progression, where proofs in Category I do not benefit from parallel execution and proofs in Category IV benefit the most. The characteristics that lead us to these categories are: the duration of a proof, how long a proof takes to arrive at a case split, and how much a proof's critical path dominates its execution time.

ACL2(p) provides both *early feedback on failure* and *faster execution* for proofs. Proofs from categories I and II do not benefit from either of these, Category III proofs benefit from early feedback on failure, and Category IV proofs benefit from faster execution and therefore also benefit from early feedback on failure.

Category I. *Short-lived proofs*: little time is required

Category II. *Mostly-linear proofs, possibly with late case splitting*: the proof is nearly complete before any case splitting occurs

Category III. *Mostly-linear proofs with early case splitting*: early case splitting provides the opportunity for early feedback about failed subgoals, but most of the proof's CPU time is attributable to a critical path of subgoals derived from the original goal

Category IV. *Proofs with time-consuming and independent subgoals*: parallelism can reduce the time required to complete a proof attempt and also provide early feedback for failed subgoals

The boundaries separating the categories are subjective. For example, for a proof to be eligible for Category IV, we require that its critical path take less than half of the proof's total processing time.

5.1 Examples of Each Category

We now provide examples for each of the above categories.

Category I: Short-Lived Proofs. Parallel execution is useless for proof attempts that complete very quickly. For example, we have used ACL2 Version 6.0 to prove the associativity of append in about 0.01 seconds. Our focus is on speeding up the user's interactive experience, so we do not target these proofs.

Category II: Mostly Linear Proofs, Possibly with Late Case Splitting. Consider the theorem named *ste-thm-weaken-strengthen*, from ACL2 regression file `workshops/1999/ste/inference.lisp` [8]. This theorem does not experience significant speedup when executing in parallel. We conclude from the proof tree shown in Figure 1 that almost all of the time is spent processing the goal that ACL2 names `Goal'6'`, after which the proof finishes almost immediately. There is no improvement in the proof's duration, and the other potential benefit from parallelism is missing: there is no early feedback for unproved subgoals.

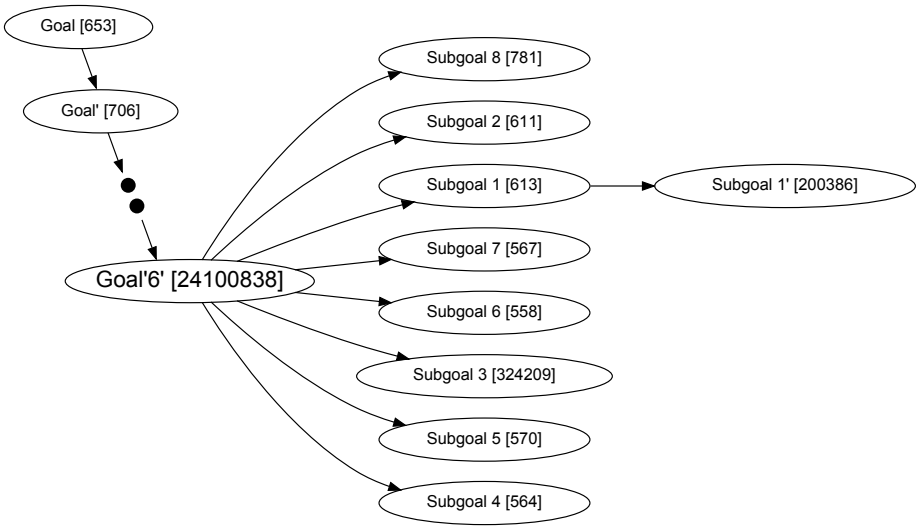


Fig. 1. Proof Dependency Tree for Theorem *Ste-thm-weaken-strengthen* (times shown in microseconds)

Category III: Mostly Linear Proofs with Early Case Splitting. We consider theorem *r-lte-r-deftraj-r-lte-r-deftrajs*, from the same file as the previous example. When executing serially, about 80% of the proof time is spent processing three dependent goals: `Subgoal *1/1'`, `Subgoal *1/1.15`, and `Subgoal *1/1.15'`, as shown in the proof dependency tree in Figure 2. Thus, there is little opportunity for speeding up the total proof time by using subgoal-level parallelism. But with parallelism enabled, if `Subgoal *1/2` or `Subgoal *1/2'` had generated a key checkpoint, it would have been immediately available to the user, instead of requiring the user to wait for other subgoals to complete[1].

Category IV: Proofs with Time-Consuming and Independent Subgoals. The opportunity for parallelism is clearly greatest for proofs that take nontrivial time and contain many independent subgoals. Here we discuss two proofs that demonstrate the ability of our system to scale and a third proof that illustrates a weakness in our implementation: (1) the proof of a theorem designed to set the baseline for the speedup we can hope to obtain on our implementation and test machine, (2) a proof about the JVM that takes a long time and has many case splits, and (3) a proof that exhibits the case where the critical path becomes stuck in the work queue.

[1] ACL2 users will notice that a sequential proof would actually have reached these checkpoints before `Subgoal *1/1'` and its descendents, but it is easy to imagine a similar example with the subgoals of `*1` reversed.

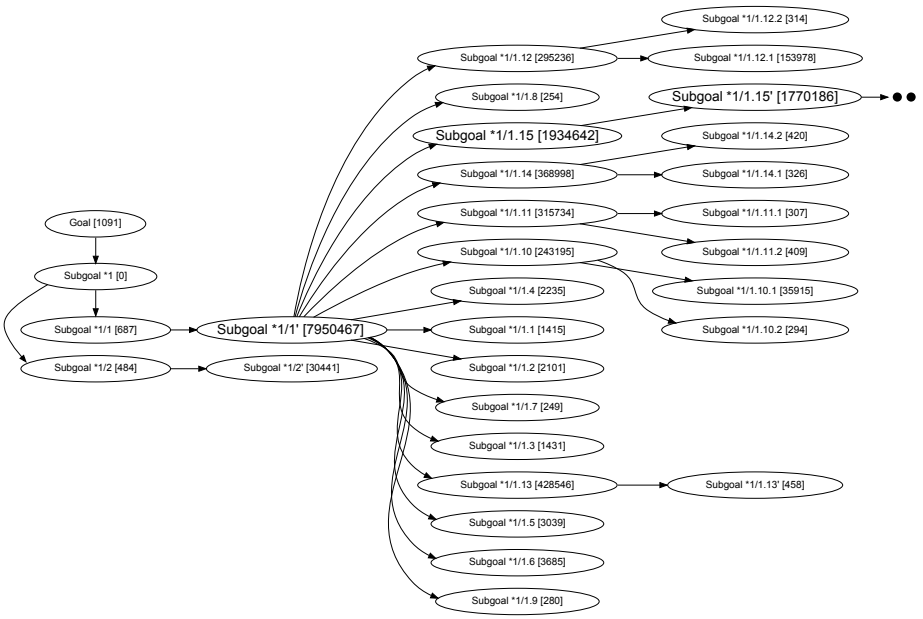


Fig. 2. Proof Dependency Tree for Theorem *R-lte-r-deftraj-r-lte-r-deftrajs* (times shown in microseconds)

Case Study: Theorem Ideal-32-way. The proof of *ideal-32-way* involves 32 distinct function calls, each counting down from a very large number and testing that the value returned is not equal to a particular constant. The prover completes each of the thirty-two generated subgoals by executing these functions, while spending only negligible time in other parts of the proof process. This proof illustrates the best speedup that we can hope to achieve with our implementation and machine. *Ideal-32-way* obtains a speedup of 25.42x on the 32-core machine. As such, it would be surprising if any proof were to obtain speedup significantly greater than 25.42x in these results.

Case Study: JVM Theorem [2b]. The proof of theorem *[2b]*, from ACL2 regression file `models/jvm/m5/apprentice.lisp` [8], is an example of a time-consuming proof that has many subgoals (2,385) and a relatively short critical path. Indeed this proof has the potential to speed up by a factor of 240.27x, and it obtains a speedup of 25.60x on our test machine.

Case Study: Theorem Step2-marks-3marked-node-either-2-or-3-or-4. When the underlying system has a large number of CPU cores, it is often the case that the work queue is quickly emptied of all pending subgoals. However, when the system has a smaller number of CPU cores (perhaps four) the work queue often stays non-empty for nontrivial durations. This sometimes results in letting the critical path of a proof sit idly in the work queue, when it would be better to prioritize

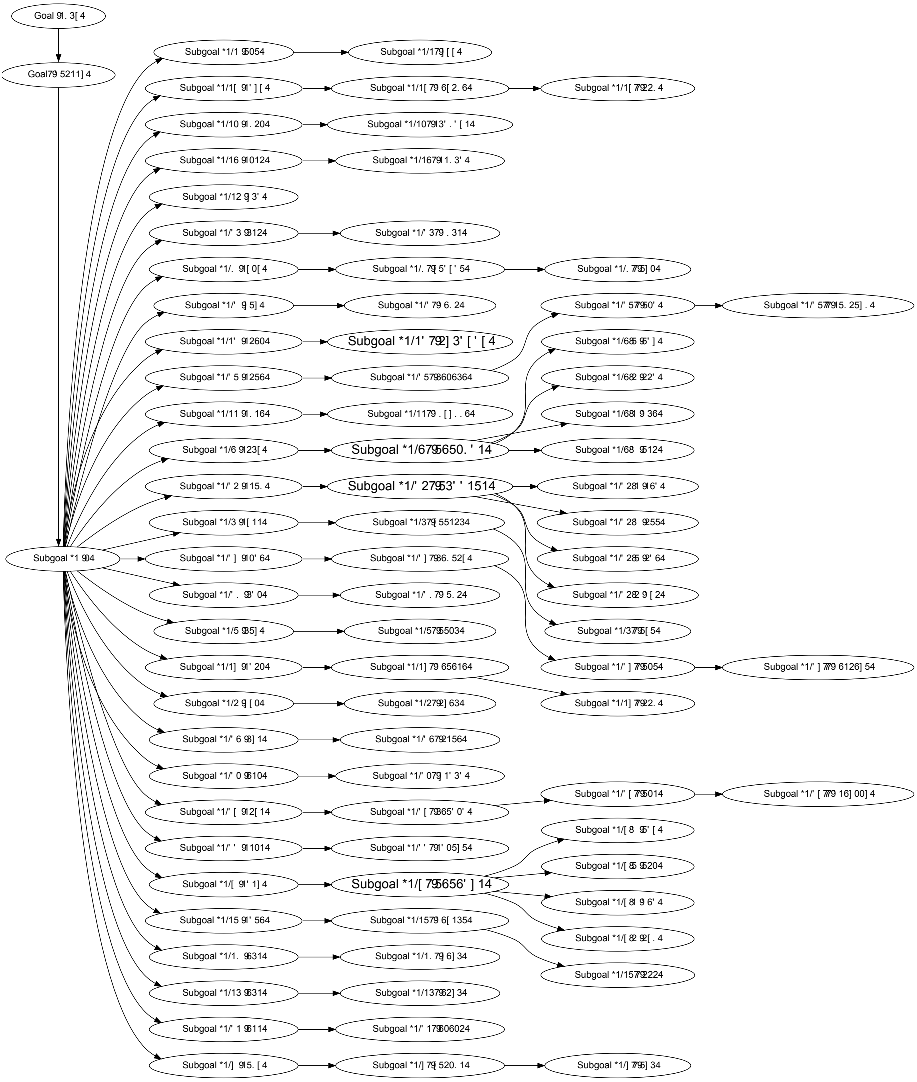


Fig. 3. Proof Dependency Tree for Theorem *Step2-marks-3marked-node-either-2-or-3-or-4* (times shown in microseconds)

the critical path and keep executing it. This lack of prioritization delays the proof attempt from finishing. We can witness the performance effects of letting the critical path sit unprocessed in the work queue by examining the proof of theorem *step2-marks-3marked-node-either-2-or-3-or-4*, from ACL2 regression file `workshops/2011/verbeek-schmaltz/sources/correctness.lisp` [8].

Figure 3 illustrates the dependencies between each subgoal of this proof. Note that `Subgoal *1/23'`, `Subgoal *1/12'`, `Subgoal *1/9'`, and `Subgoal *1/5'` take more time than the others. As such, it would make sense to start each of these subgoals as soon as they become available. In practice, however, this does not occur. On a machine with eight cores, the start of `Subgoal *1/5'` is always delayed [12]. This results in an observed experimental speedup that is significantly less than the potential speedup that the proof could obtain if we prioritized `Subgoal *1/5'`.

Indeed, as shown in Figure 5, many proofs experience far from optimal speedup, and we hypothesize that this lack of prioritization is part of the cause. It would be ideal to prioritize the critical path, but predicting which path is critical is a known difficult problem ([33], Section 5.9.3). With this in mind, we have a mode that uses timing information from prior executions, but it is only in an experimental state and we leave its further development and investigation of this issue as future work.

6 Experimental Results

In this section we present timing information and corresponding analysis, indicating that we have successfully sped up the execution time for many proofs. Of course, these proofs all succeed since they are part of the regression suite. But successful parallelization can also lead to quicker feedback upon failure.

Our test machine has four Intel E5-4650 8-core processors and 128 gigabytes of RAM. Hyper-threading adds relatively little ([12], Section 7.3.2) and is disabled. This yields a total of 32 cores and 32 hardware threads. We measure proofs from the ACL2 regression suite, omitting sets of files with features that are inherently single-threaded and using a version of ACL2 6.0 that we only modified to save raw performance data [9]. The numbers reported in this section are averages of at least four runs of each test.

We use the default parallelism mode, *resource-based*, for measuring the time it takes to run the prover in parallel, and we use a serial mode named *pseudo-parallel* to measure the time it takes to run the prover serially. The pseudo-parallel mode is a good choice for comparison because it has the same code base as the resource-based mode, with one exception: whenever the system asks whether to parallelize execution, it opts to continue serially. Nevertheless, a comparison with ACL2 instead of with the pseudo-parallel mode in ACL2(p) would have yielded slightly different results: the time measured for a test run [9] using ACL2 was 136.59 minutes, and the corresponding time measured using ACL2(p)'s pseudo-parallel mode was 153.14 minutes; thus, ACL2 takes approximately 11% less time than the pseudo-parallel mode. For a comparison against

(non-parallel) ACL2, one should multiply the reported speedup results by a factor of 0.89.

We run all of our tests with a garbage collection threshold of 96 gigabytes. Since garbage collectors are single-threaded in our host Lisp implementations, one might wonder whether having a smaller garbage collection threshold (as might be necessary on a laptop) would cause poorer speedup. Indeed, when we run the proof of theorem *[2b]* with a 2 gigabyte threshold on a machine with only 8 cores, the speedup decreases from 7.48x to 6.65x [12]. This decrease suggests that running our tests with a lower garbage collection threshold would reduce performance, though the reduction would be modest.

6.1 Defining Our Metrics

A crucial determinant of the performance improvement possible with our parallelization of a given proof is its *potential speedup*. We define the potential speedup for a proof to be the quotient of the sequential proof time on a given machine divided by the time required to complete the proof's critical path. If, for example, a proof's critical path completes in 10 seconds and the entire proof completes in 21 seconds, then the potential speedup is 21/10, or 2.1x. The *grade* for a given run is then defined to be the theorem's observed speedup divided by the theorem's potential speedup *for a particular machine*. By this we mean that the denominator of a grade is the minimum of the potential speedup and the number of cores available on the machine that generated that observed speedup. Consider the following two examples running on an 8-core machine: if a proof's potential speedup is 100x, then an observed speedup of 4x results in a 50% grade; while if a proof's potential speedup is 2x, then an observed speedup of 1.8x results in a 90% grade.

6.2 Performance Results

Table 1 shows the observed speedup, potential speedup, and calculated grade for each of the twenty-five proofs measured to have the longest execution times on our test machine. This table also includes our category labels for each theorem. Of these theorems, we label twenty as *Category IV*. This implies that many lengthy proof attempts can benefit from parallel execution in both execution time and early feedback. Of the remaining five proofs, three of them fall into *Category III*, and two of them fall into *Category II*. Thus, only 8% of this sample fails to benefit from parallel execution, suggesting that proof attempts that take nontrivial time can generally benefit from parallel execution.

Many of these twenty-five proofs experience a speedup that is quite useful. In particular, of these twenty-five proofs, eight of them obtain a speedup in excess of 10x. Although the proof of theorem *[2a]* has a potential speedup much larger than 32x, it is perhaps surprising that its speedup falls short of *ideal-32-way*'s speedup of 25.42x. Our study of theorem *step2-marks-3marked-node-either-2-or-3-or-4*, among others, provides a plausible explanation: the critical path is left stuck in the work queue. Future work may attempt to affirm this hypothesis.

Table 1. Performance Improvement of Twenty-Five Longest Running Theorems

Theorem	Obs SU	Pote SU	Grade	Ser Time	Par Time	Cat
[2b]	25.60	240.27	80%	411.59	16.08	IV
[3b]	25.78	151.94	81%	225.17	8.74	IV
dlf->not3	1.63	1.64	99%	224.02	137.57	II
spec-body	17.22	21.86	79%	141.30	8.21	IV
step1-puts-dest-to-neighb...	6.21	6.64	94%	121.04	19.48	IV
step1-puts-all-neighbors-...	4.72	4.98	95%	105.28	22.31	IV
step1-puts-all-neighbors-...	4.50	4.77	94%	94.04	20.90	IV
step1-preserves-dl->not2-...	4.06	4.27	95%	92.24	22.71	IV
step1-puts-all-neighbors-...	4.41	4.67	94%	90.56	20.55	IV
[2a]	20.05	50.95	63%	87.64	4.37	IV
step1-preserves-invariant...	5.23	5.69	92%	82.91	15.84	IV
ub-g-chain-=-g-chain-skol...	10.41	12.46	84%	80.33	7.71	IV
fw	1.95	1.96	99%	71.43	36.67	III
step1-gives-0marked-node-...	3.60	3.66	98%	66.94	18.60	IV
temp14.00	3.03	3.17	96%	60.72	20.02	IV
convert-normalized-term-t...	19.67	73.92	61%	57.36	2.92	IV
[3a]	21.21	36.51	66%	51.41	2.42	IV
cases-on-th	24.31	65.80	76%	49.71	2.04	IV
lemma5-for-utf8-combine4-...	1.00	1.00	100%	48.56	48.46	II
lemma1_last_route_is_to	2.73	2.81	97%	47.70	17.49	IV
simple-genoc-is-correct	1.28	1.29	99%	46.65	36.35	III
wp-zcoef-g=h	1.71	1.74	98%	46.42	27.18	III
inside-universalp-step	7.59	8.92	85%	45.91	6.05	IV
step1-preserves-invariant...	3.30	3.34	99%	42.57	12.89	IV
equal-wp-zcoef-g	4.61	5.08	91%	39.84	8.63	IV

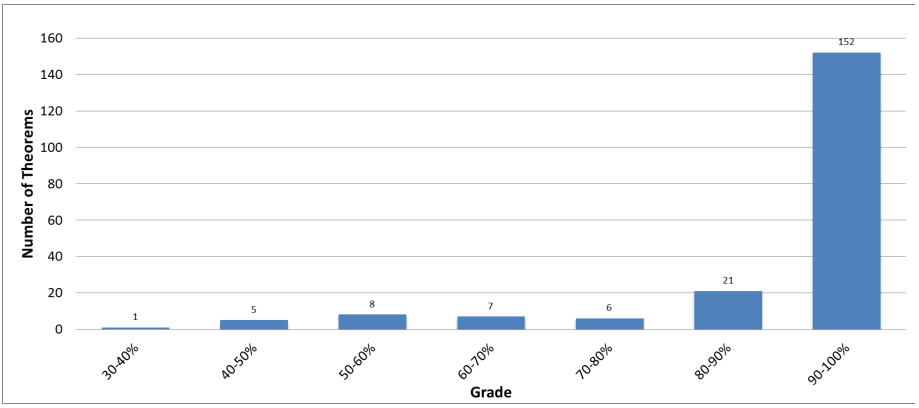


Fig. 4. Number of Theorems (of the top 200 longest running theorems) for Each Range of Grade

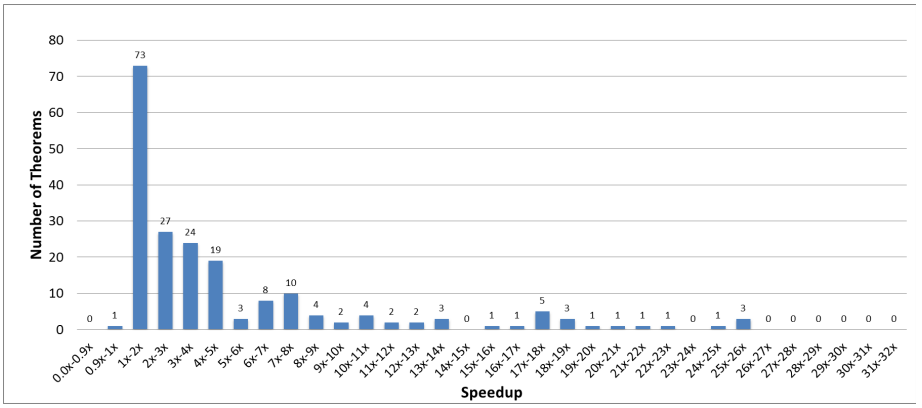


Fig. 5. Number of Theorems (of the top 200 longest running theorems) with Given Observed Speedup

In Figure 4, we group the two hundred longest proofs into batches based on their grade. As shown in the figure, 152 theorems achieve a grade of 90% or higher, 21 achieve a grade between 80% and 90%, and 27 achieve a grade less than 80%. The theorems at the lowest tiers function as starting points for improving the efficiency of our subgoal-level parallelism implementation in the future. Figure 5 groups the same 200 proofs into batches based on their observed speedup. While it is good that many of the theorems obtain nontrivial speedup, it is equally noteworthy that no theorem slows down by more than 10% of its serial time (known because there are no theorems in the column labeled "0.0x - 0.9x"). The theorems with nontrivial proof durations that obtain little speedup provide a foundation for researching alternatives to subgoal-level parallelism.

We discuss the speedup achieved by our system because speedup, or speedup per core, is a common metric for success when parallelizing software. However, focusing exclusively on speedup undermines a more important result: how much time we are able to save the user. Take, for example, the proof of theorem *[2b]*. This proof takes 412 seconds when executing serially on our state-of-the-art machine. Needless to say, it takes even more time when running on older, less well-equipped systems [12]. However, when users enable parallel execution for this proof, they only need to wait 16 seconds for the proof to complete. This time difference is qualitatively significant, as it can help a user to avoid switching contexts or, as we discuss in the conclusion, building theories to avoid case splits.

It is encouraging that speedup has occurred in so many proofs from the ACL2 regression suite, which were developed when it was useful to spend effort to minimize large case splits that now could be sped up by ACL2(p). As we look forward, we expect that users will avoid expending that effort, choosing instead to benefit from the parallel execution that ACL2(p) provides.

7 Conclusion and Future Work

The introduction of parallel execution into the proof process improves a user's interactive experience with the ACL2 prover. In essence, we have used the ACL2 language, extended with parallelism features, to parallelize a very large program — the ACL2 prover — providing significant speedup for interactive use. This effort required thorough study of the ACL2 theorem-proving process and the careful introduction of parallelism into that process; our early attempts provided insufficient benefit.

Our strategy for parallelization of the ACL2 theorem-proving process involved: improving the mechanisms necessary to execute ACL2 programs robustly in parallel; using these mechanisms to parallelize the main proof process of ACL2, while continuing to provide useful (and early) feedback to users; and designing and analyzing experimental results that confirm the benefits of parallel execution for automated proof. The resulting system, ACL2(p), can be built by downloading ACL2 and using a compile-time switch. It is thus available for development of parallel ACL2 programs, and it is used to speed up the development and replaying of proofs.

A parallel proof environment on a modern multi-core machine also offers the opportunity to initiate concurrent proof attempts automatically on a given goal, using different proof strategies, and without much penalty. Future work may pursue this opportunity, for example by parallelizing ACL2's *or-hints* [5] mechanism. Future work may also improve the parallelism mode that uses timing information from a theorem's proof attempt to prioritize the critical path during that theorem's subsequent proof attempts.

As parallel proof becomes more common, users will discover that their interactive experience improves in qualitative ways beyond faster execution and early feedback. As an example, ACL2 users have often put significant effort into managing their theories to avoid large case splits, by proving additional theorems that help guide the ACL2 prover to proofs with fewer cases. For subgoals generated from case splits, much of this proof-engineering effort becomes unnecessary. The benefit goes beyond shortening the user's wait for a proof — the user may be able to avoid development of lemmas that prevent case splits. Thus, the result of our work is not just a change in performance; it supports a paradigm shift in how users interact with a mechanical theorem-proving system.

Acknowledgements. This material is based upon work supported by The Battelle Memorial Institute, by DARPA under Contract No. N66001-10-2-4087, by ForrestHunt, Inc., and by the National Science Foundation under Grant Nos. CCF-0945316 and CNS-0910913. We thank Robert Krug and Shilpi Goel for being early adopters of ACL2(p) and Jared Davis, Sung Jun Lim, J Strother Moore, and Nathan Wetzler for numerous helpful discussions. Finally, we thank the reviewers for helpful comments.

References

1. Kaufmann, M., Manolios, P., Moore, J. S : Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston (2000)
2. Brock, B., Kaufmann, M., Moore, J. S : ACL2 theorems about commercial microprocessors. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 275–293. Springer, Heidelberg (1996)
3. Russinoff, D., Kaufmann, M., Smith, E., Sumners, R.: Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. In: Nikolai, S. (ed.) Proceedings of the 17th IMACS World Congrress on Scientific Computation, Applied Mathematics and Simulation, Paris, France (2005)
4. Hunt Jr., W.A., Swords, S., Davis, J., Slobodova, A.: Use of formal verification at Centaur Technology. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 65–88. Springer, US (2010)
5. ACL2: ACL2 Version 6.0 Documentation (December 2012), `http://www.cs.utexas.edu/users/moore/acl2/v6-0/acl2-doc.html#User's-Manual`
6. Rager, D.L.: Implementing a parallelism library for ACL2 in modern day Lisps. Master's thesis, The University of Texas at Austin (2008)
7. Rager, D.L., Hunt Jr., W.A.: Implementing a parallelism library for a functional subset of Lisp. In: Proceedings of the 2009 International Lisp Conference, pp. 18–30. Association of Lisp Users, Sterling (2009)
8. ACL2 Community Books, `https://code.google.com/p/acl2-books/`
9. Rager, D.L.: ACL2 6.0 regression suite test configuration details, `http://www.cs.utexas.edu/users/ragerdl/papers/itp2013/`
10. Moore, J. S , Porter, G.: The apprentice challenge. ACM Transactions on Programming Languages and Systems 24, 193–216 (2002)
11. Verbeek, F., Schmaltz, J.: Formal verification of a deadlock detection algorithm. In: Hardin, D., Schmaltz, J. (eds.) Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, November 3-4. Electronic Proceedings in Theoretical Computer Science, vol. 70, pp. 103–112. Open Publishing Association (2011)
12. Rager, D.L.: Parallelizing an Interactive Theorem Prover: Functional Programming and Proofs with ACL2. PhD thesis, The University of Texas at Austin (2012)
13. Moten, R.: Exploiting parallelism in interactive theorem provers. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 315–330. Springer, Heidelberg (1998)
14. Meseguer, J., Winkler, T.C.: Parallel programmming in Maude. In: Banâtre, J.-P., Le Métayer, D. (eds.) Research Directions in High-Level Parallel Programming Languages 1991. LNCS, vol. 574, pp. 253–293. Springer, Heidelberg (1992)
15. Bonacina, M.P., McCune, W.: Distributed theorem proving by peers. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 841–845. Springer, Heidelberg (1994)
16. Kapur, D., Vandevoorde, M.T.: DLP: a paradigm for parallel interactive theorem proving (1996)
17. Schumann, J., Letz, R.: PARTHEO: A high-performance parallel theorem prover. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 40–56. Springer, Heidelberg (1990)
18. Schumann, J.: SicoTHEO: Simple competitive parallel theorem provers. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 240–244. Springer, Heidelberg (1996)

19. Matthews, D.C.J., Wenzel, M.: Efficient parallel programming in poly/ML and isabelle/ML. In: DAMP 2010: Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, pp. 53–62. ACM, New York (2010)
20. Wenzel, M.: Parallel proof checking in Isabelle/Isar. In: Reis, G.D., Théry, L. (eds.) ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS). ACM Digital library (August 2009)
21. Wenzel, M.: Shared-memory multiprocessing for interactive theorem proving. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Fourth Conference on Interactive Theorem Proving (ITP 2013). LNCS, vol. 7998, pp. 414–429. Springer, Heidelberg (2013)
22. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Academic Press, New York (1988)
23. Kaufmann, M., Wilding, M.: A parallel version of the Boyer-Moore prover. Technical Report 39. Computational Logic, Inc. (February 1989)
24. Halstead Jr., R.H.: Implementation of multilisp: Lisp on a microprocessor. In: Conference on Lisp and Functional Programming, pp. 9–17 (1984)
25. Goldman, R., Gabriel, R.P., Sexton, C.: Qlisp: An interim report. In: Ito, T., Halstead Jr., R.H. (eds.) US/Japan WS 1989. LNCS, vol. 441, pp. 161–181. Springer, Heidelberg (1990)
26. Harrison, W.L.: The interprocedural analysis and automatic parallelization of scheme programs. Lisp and Symbolic Computation 2(3), 179–396 (1989)
27. Harrison, W.L., Ammarguellat, Z.: A comparison of automatic versus manual parallelization of the Boyer-Moore theorem prover. In: Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing, pp. 307–330. Pitman Publishing, London (1990)
28. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
29. University of Cambridge: HOL4 Kananaskis 5 (March 2010), `http://hol.-sourceforge.net/`
30. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
31. SRI International: PVS Specification and Verification System (July 2012), `http://pvs.csl.sri.com/`
32. Rager, D.L., Hunt Jr., W.A., Kaufmann, M.: A futures library and parallelism abstractions for a functional subset of Lisp. In: Proceedings of the 4th European Lisp Symposium (March 2011)
33. Schumann, J.: Automated theorem proving in software engineering. Springer (2001)

Communicating Formal Proofs: The Case of Flyspeck

Carst Tankink[1], Cezary Kaliszyk[2], Josef Urban[1], and Herman Geuvers[1,3]

[1] ICIS, Radboud Universiteit Nijmegen, Netherlands
[2] Institut für Informatik, Universität Innsbruck, Austria
[3] Technical University Eindhoven, Netherlands

Abstract. We introduce a platform for presenting and cross-linking formal and informal proof developments together. The platform supports writing natural language 'narratives' that include islands of formal text. The formal text contains hyperlinks and gives on-demand state information at every proof step. We argue that such a system significantly lowers the threshold for understanding formal development and facilitates collaboration on informal and formal parts of large developments. As an example, we show the Flyspeck formal development (in HOL Light) and the Flyspeck informal mathematical text as a narrative linked to the formal development. To make this possible, we use the Agora system, a MathWiki platform developed at Nijmegen which has so far mainly been used with the Coq theorem prover: we show that the system itself is generic and easily adapted to the HOL Light case.

1 Introduction

Formal proof development is gradually becoming accepted as a means for establishing the correctness of mathematical theory in particular. Large repositories of formal proof have been created in various proof assistants to prove impressive results, for example, the development of the odd order theorem in Coq [1], the proof of the 4 color theorem in Coq [2] and the proof of the Kepler conjecture [3] in HOL Light. A major issue is how to communicate these large formalizations: to people that want to cooperate or want to build further on the development, to people who want to understand the precise choices (of definitions and proofs) chosen in the formalization and to people who want to convince themselves that it is really the proper theorem that has been proven. At the moment, communicating a formal proof is hard, as can also be noticed from the fact that the number of publications about the impressive formalizations mentioned above is low. Moreover, these publications hardly give access to the formalization, but describe the project on a rather high level of abstraction. The Journal of Formalized Reasoning[1], the Archive of Formal Proofs[2] and the Journal of Formalized

[1] `http://jfr.unibo.it/`
[2] `http://afp.sf.net`

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 451–456, 2013.

Mathematics[3] try to improve on this by explicitly giving a platform for formalizations (the latter for Mizar), but that is not really taking off.

In the present paper we present a wiki based approach towards the communication of large formal proof developments. Formal proofs are close to programs written in a high-level programming language, which need to be documented to be understandable and maintainable. However, a mathematical proof development is also special, because there (almost always) already is documentation, which is the mathematical document (a book or an article) describing the mathematics (definitions, notation, lemmas, proofs, examples).[4] This is what we call *informal mathematics* as opposed to *formal mathematics* which is the mathematics as it lives inside a proof assistant. These days, informal mathematics consists of LaTeX files and formal mathematics usually consist of a set of text files that are given as input to a proof assistant to be checked for correctness. Our approach is to provide tools that allow users to do the following.

1. One can automatically generate wiki files from formal proof developments. These wiki files can then be displayed in a browser, where we maintain all linking that is inherently available in the formal development (e.g. via definitions and applications of lemmas).
2. When hovering over the formal proofs, one sees the proof state at that point, so a reader can observe what the action of the proof commands is. This uses the Proviola technology that we have previously developed and described [4].
3. One can also automatically generate wiki files from a set of LaTeX files. These wiki files can then be displayed in a browser, where we maintain the linking inside the LaTeX files, but more importantly, also the linking with the formal proof development.
4. One can write a wiki document about mathematics and include snippets of formal proof text via an inclusion mechanism. This allows one to dynamically insert a piece of formal proof, by referencing the formal object in a repository, which is then automatically rendered and displayed inside the wiki document.

The tools we describe are part of the Agora system we are developing in Nijmegen, which aims at being a "Wiki for Formal Mathematics": a web platform to present and document formalizations, but also to cooperate on joint formalizations. With Agora, we want to lower the threshold for participating in formalization projects by:

- Providing an easy-to-use web interface to a proof assistant [5].
- Marking up formal mathematics for the Web without effort by authors [6], allowing users to browse this database for examples and inspiration.

[3] `http://fm.mizar.org/`

[4] Formalizations of (software) systems typically have an informal *specification*, which serves a similar role as a mathematical document, and which could be served by the tools described in this paper, although some of the workflows described here might not match completely.

– Providing tools for linking informal and formal text [7].
– Providing additional tools for users of proof assistants, like automation or proof advice.

The system is designed to support the dissemination of formal mathematics to an audience that does not necessarily have prior exposure to an interactive theorem prover. Our general claim is that this type of technology is crucial to further the field of formalized mathematics. One has to develop computer support for documenting and communicating formal proofs and for linking formal proofs to a high level 'narrative'.

Until recently, the system only fully supported formalizations written for the Coq theorem prover, having been tested on smaller test cases. However, the system is designed to be generic, reusing components that can be specialized for specific theorem provers. This paper describes the extension of Agora to include the HOL Light theorem prover [8], to allow the system to serve the files of the Flyspeck project in a wiki.

2 Presenting Flyspeck in Agora

In the present paper we will not go into the general goals or design of Agora, but only show the tools that support the 4 activities mentioned above. We show the practical usability of the tools by presenting a page of the Flyspeck formal development in HOL Light, together with the page of the informal mathematical description (Figure 1). By discussing these pages, the links between them and how they have been created, we describe our tools.

An example document resides in Agora,[5] and is shown in part in Figure 1. For the best experience, we suggest the reader follows along at the demonstration page while reading this section. Implementing low-threshold interactive web-editing of formal HOL Light code is currently work in progress.

2.1 Description of a Formal Proof

The first noticeable feature of the document is that it is almost isomorphic to Chapter 5 of the text accompanying the Flyspeck formalization [9] (our source document). As mentioned above, important formalizations are not merely technical proof scripts: they go hand-in-hand with informal (in this case Hales's) mathematical narrative. To obtain the informal text, we have processed the LaTeX sources of Hales's text, transforming it into the Creole syntax [10]. This syntax is similar to Wikipedia's input language: a light-weight markup language that is easy to translate to HTML. The formulae in the source document are kept largely intact: they are processed at render-time by MathJax[6]: a JavaScript tool for rendering mathematics in a browser-independent way. This approach makes the resulting document editable as a wiki page written in Creole. A more complete approach would be to also accept LaTeX as input language for writing the documentation, something we intend to address as a follow-up.

[5] http://mws.cs.ru.nl/agora_flyspeck/flyspeck/fly_demo
[6] http://mathjax.org

Lemma [node partition] VBTIKLP (disjoint) [edit]

Let (V, E) be a fan. Let $\mathbf{v} \in V$. Then a disjoint sum decomposition of $\mathbb{R}^3$ is given by

$$\mathbb{R}^3 = \mathrm{aff}\{\mathbf{0}, \mathbf{v}\} \cup \bigcup_{\mathrm{node}(x)=\mathbf{v}} W^0_{\mathrm{dart}}(x) \cup \bigcup_{\{\mathbf{v},\mathbf{w}\}\in E} \mathrm{aff}^0_+(\{\mathbf{0}, \mathbf{v}\}, \mathbf{w}).$$

Proof [edit]

We start the proof with the existence of the disjoint sum decomposition. First of all, $\mathbb{R}^3$ is the disjoint union of $\mathrm{aff}\{\mathbf{0}, \mathbf{v}\}$ and its complement.
The case when $\mathrm{card}(E(\mathbf{v})) \leq 1$ follows immediately from the definitions. Therefore, assume that $\mathrm{card}(E(\mathbf{v})) > 1$. Fix $\mathbf{u}$ such that $\{\mathbf{v}, \mathbf{u}\} \in E$, and let σ be the azimuth cycle on $E(\mathbf{v})$. Let $\alpha(i) = \mathrm{azim}(\mathbf{0}, \mathbf{v}, \sigma^i\mathbf{u}, \sigma^{i+1}\mathbf{u})$. By Lemma 2pi-sum , the sum of the angles $\alpha(i)$ is 2π. Every

```
let VBTIKLP=prove(`(!(x:real^3) (V:real^3->bool) (E:(real^3->bool)->bool)  (v:real^3) (u:real^3).
FAN(x,V,E)/\ ({v,u}IN E)
==>
 (UNIV:real^3->bool) = aff {x,v} UNION (UNIONS {w_dart_fan x V E (x,v,w,(sigma_fan x V E v w))|w| {v,w} IN E })
UNION
 (UNIONS {aff_gt {x,v} {w} |w| {v,w} IN E} ))
/\
(!(x:real^3) (V:real^3->bool) (E:(real^3->bool)->bool)  (v:real^3) (w:real^3).
FAN(x,V,E)/\ ({v,w}IN E)
==>
 w_dart_fan x V E (x,v,w,(sigma_fan x V E v w)) INTER aff {x,v}={})
/\
```

Fig. 1. An informal proof together with its formal counterpart. Cropped screenshots from document pages at `http://mws.cs.ru.nl/agora_flyspeck/flyspeck/fly_demo`.

2.2 Integration with Formal Proof

A nicely marked up paper, whether or not it appears as a Web page, is not a description of a formal proof: for this, it needs to include parts of the formalization, in order to showcase and document them: this inclusion does not have to be complete, as this might muddle the description with details that are not immediately necessary for understanding. So, the second feature of the document is that the definitions and lemmas in it are surrounded by a box, and marked with buttons marked "formal" and "informal": using these buttons, a reader can toggle between the informal text of such a text and the corresponding formalization.

This functionality is made possible thanks to (Hales's) annotations of the source text, combined with a previously developed technology for Agora. For (almost) each island of 'mathematics' (definitions, lemmas, theorems...), the source text defines the corresponding entity or entities in the formal development. The corresponding entity can be included in the page using Agora's inclusion facility [7], by transforming the correspondence into a kind of hyperlink. The necessary syntax can be also hand-written in the wiki (this can be used for gradual addition of more and more cross-links to the formal code), but so far everything was generated from the source text annotations.

This approach differs from Isabelle/Isar [11], which supports a user in writing a formal proof and its documentation in a more literate way: the full proof is

part of the document, and can be verified by the system. Agora's documentation tools, on the other hand, allow writing a documentation layer on top of the formal code within the system.

2.3 Dynamic Display

The Proviola tool integrated in Agora's rendering chain reduces the task of evaluating proof state to just pointing at parts of the proof: it shows a proof script as HTML[7] and when the user points at a particular command, the associated state is computed in the background, caching it for retrieval without computation at a later moment.. This interaction model has two advantages: (i) it eliminates the overhead of installing, configuring, and learning about a theorem prover; inspection of an interesting proof state or tactic is reduced to pointing a cursor at it, and (ii) it reduces the amount of task switches a reader will have to make.

Proviola has a generic design, and the inclusion of HOL Light for its batch task was simple: writing a parser that recognizes input to the prover, and some glue that allows the Proviola to send these commands to HOL Light, and read the output.

3 Conclusion and Future Work

Agora is an online platform that facilitates collaborative gradual formalization of mathematical texts, and allows their dual presentation as both informal and formal. In particular, the platform takes both LaTeX and formal input, cross-links both of them based on simple user-defined macros and on the formal syntax, and allows one to easily browse the formal counterparts of an informal text. One future direction is to allow even the non-mathematical parts of the wiki pages to be written directly with (extended) LaTeX, as it is done for example in PlanetMath. This could facilitate the presentation of the projects developed in the wiki as standalone LaTeX papers. On the other hand, it is straightforward to provide a simple script that translates the wiki syntax to LaTeX, analogously to the existing script that translates from LaTeX to wiki. We also still have to instantiate the interactive editing capability (now available for Coq) to HOL Light. This editing capability would allow a reader to directly edit the formal islands in the wiki pages by talking to a server-side HOL Light instance (with a reasonably advanced checkpointed state), loaded with additional prerequisities (in particular the previous formal islands). Allowing 'anyone' to edit does raise questions about how to maintain the integrity of a formalization. While we do not have an implemented solution to this in Agora, there are several options available [12,13] to deal with these issues, which still need to be evaluated on usability: an ideal solution needs to prevent a user thrashing the library, but does not shut out new users by overly complex or time-consuming protocols.

We are working on integrating the recently developed proof advice system [14] for HOL Light. The advisor uses machine learning to find lemmas that can be useful in solving a goal, encodes the goal together with the advised lemmas in TPTP

[7] E.g., `http://mws.cs.ru.nl/agora_flyspeck/flyspeck/fan/fan_misc/index`

format and runs a number of ATPs to find a minimal set of needed lemmas to then reconstruct the goal in HOL. This can be especially useful in a Wiki environment with many (possibly non-expert) contributing users, where it can be also used to automatically discover redundancies and refactor the formalization. Another direction is adding good linguistic techniques for translating informal texts to formal ones based on training on the annotated corpora (arising through this work). We could also try to include (or even better: track) informal wikis like ProofWiki, and start adding formal counterparts and annotations to them. Similarly for papers and books that were formalized in various systems: for example the books leading to the proof of Feit-Thomson theorem and their recent Coq formalization, and the Compendium of Continuous Lattices and its Mizar formalization.

References

1. Gonthier, G.: Engineering mathematics: the odd order theorem proof. In: Giacobazzi, R., Cousot, R. (eds.) POPL, pp. 1–2. ACM (2013)
2. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: Kapur, D. (ed.) ASCM 2007. LNCS (LNAI), vol. 5081, pp. 333–333. Springer, Heidelberg (2008)
3. Hales, T.C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., Zumkeller, R.: A revision of the proof of the Kepler conjecture. Discrete & Computational Geometry 44(1), 1–34 (2010)
4. Tankink, C., Geuvers, H., McKinna, J., Wiedijk, F.: Proviola: A tool for proof re-animation. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 440–454. Springer, Heidelberg (2010)
5. Tankink, C.: Proof in context — Web editing with rich, modeless contextual feedback. To appear in Proceedings of UITP 2012 (2012)
6. Tankink, C., McKinna, J.: Dynamic proof pages. In: ITP Workshop on Mathematical Wikis (MathWikis). Number 767 in CEUR Workshop Proceedings (2011)
7. Tankink, C., Lange, C., Urban, J.: Point-and-write – documenting formal mathematics by reference. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS, vol. 7362, pp. 169–185. Springer, Heidelberg (2012)
8. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009)
9. Hales, T.C.: Dense Sphere Packings - a blueprint for formal proofs. Cambridge University Press (2012)
10. Sauer, C., Smith, C., Benz, T.: Wikicreole: A common wiki markup. In: WikiSym 2007, pp. 131–142. ACM, New York (2007)
11. Wenzel, M., Paulson, L.: Isabelle/isar. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. LNCS (LNAI), vol. 3600, pp. 41–49. Springer, Heidelberg (2006)
12. Alama, J., Brink, K., Mamane, L., Urban, J.: Large formal wikis: Issues and solutions. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) Calculemus/MKM 2011. LNCS, vol. 6824, pp. 133–148. Springer, Heidelberg (2011)
13. Corbineau, P., Kaliszyk, C.: Cooperative repositories for formal proofs. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 221–234. Springer, Heidelberg (2007)
14. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. CoRR abs/1211.7012 (2012)

Square Root and Division Elimination in PVS

Pierre Neron

École polytechnique - Inria

Abstract. In this paper we present a new strategy for PVS that implements a square root and division elimination in order to use automatic arithmetic strategies that were not able to deal with these operations in the first place. This strategy relies on a PVS formalization of the square root and division elimination and deep embedding of PVS expressions inside PVS. Therefore using computational reflection and symbolic computation we are able to automatically transform expressions into division and square root free ones before using these decision procedures.

Introduction. Proof verification systems such as PVS [7] embed proofs strategies that allow the user to deal with arithmetic problems automatically. However most of these techniques such as the use of SMT solvers [2,4] or quantifier elimination [3] are not able to manage all arithmetics operations, in particular division and mainly square roots. Being able to transform any goal or hypothesis containing square roots or divisions into an equivalent one that is free of them would allow the use of arithmetic decision procedures to resolve the current goal.

A program transformation that removes square roots and divisions from programs has been defined and proved correct in PVS, see [6]. We now aim at using this implementation of the transformation and the proof of the semantics equivalence between the input and the output formulas to define a PVS strategy [1]. This strategy, `elim-sqrt`, transforms any goal or hypothesis by eliminating square roots and divisions from it *e.g.,*

```
{-1} x <= 1                          {-1} x <= 1
  |----                    ⟶          |----
{1} x <= sqrt(x)        elim-sqrt    {1} x * x - x <= 0
```

This is realized by doing a deep embedding [8] of a fragment of PVS inside PVS in order to use computational reflection for transformation computation [5]. This is a big difference with PVS or Coq `fields` strategies, that are written in the strategy language, since the size of the proof does not depend on the input terms.

1 Deep Embedding

First of all we need to sketch how this transformation is specified in PVS, the complete definition can be found in [6].

Definitions. The transformation in PVS is defined on programs represented in an abstract datatype `program`. It represents variables, constants, some operators, pairs, projections, variable definitions and conditional expressions:

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 457–462, 2013.

Definition 1.1 (`program` abstract datatype)

```
program : DATATYPE
        value(va : variable) : value?
        const(co : constant) : const?
        uop(uop : unop, pr : program) : uop?
        bop(bop : binop, pl : program, pr : program) : bop?
        pair( pl : program, pr : program) : pair?
        fst( pr : program) : fst?
        snd( pr : program) : snd?
        letin( x : variable, body : program, scope : program) : letin?
        ift(fm : program, prt : program, prf : program) : ift?
```

The operators in the `binop` and `unop` datatypes represent $+, -, *, /, >, \geq, =, \vee, \wedge$ and $\neg$. Functions computing the type and the semantics of a `program` in a given environment are defined in PVS. The semantics of a `program` is either a failure (*e.g.,* division by 0) or a tuple of boolean and numerical values:

Definition 1.2 (Semantics function)

```
sem(p : program, env : eval_env) : RECURSIVE prog_val
```

where `prog_val : DATATYPE`

```
            numv(re : real): numv?
            boolv(bo : bool): boolv?
            pairv(vl : prog_value, vr : prog_value): pairv?
            failv: failv?
```

and `eval_env = [variable -> prog_val]`

Given these definitions, we can now introduce the main definition of the transformation as a PVS function `elim` defined on `program`. PVS subtyping allows us to embed the preservation of the semantics in the type of this function:

Definition 1.3 (Main transformation)

```
elim(p : program) :
    {pp : program_N_sq | preserves_semantics_no_fail(p)(pp)}
```

where `program_N_sq` *is the subtype of* `program` *without square roots and divisions and* `preserves_semantics_no_fail(p)(pp)` *the following statement:*

$\forall$ `env, nofailv(sem(p,env)) IMPLIES sem(p,env) = sem(pp,env)`

In order to use this transformation, we have to transpose a PVS statement into this formalism, this realizes a deep embedding of a fragment of PVS inside PVS.

Deep Embedding. Given a proof context in PVS, we aim at transforming a statement (either a goal or an hypothesis) into an equivalent one which is free of divisions and square roots. First of all, as we can see in definition 1.1 the formalism only represents a fragment of PVS, therefore the statement we want to transform has to match this formalism. Given such a statement, we call it `S`,

the first step of this embedding is to compute the equivalent `p : program` and the corresponding evaluation environment `env` such that:

```
sem(p,env) = boolv(S)
```

Indeed, the `variable` of the `program` type are not PVS variables but identifiers (*e.g.,* string or natural numbers), therefore we need the environment to make the link between these identifiers and their value, *i.e.,* the value of the corresponding PVS variables. From now on, given a PVS variable `x` in a statement and the corresponding environment `env`, its identifier will be the string `"X"`. These elements, the `program` and `environment`, have to be computed as their PVS string representation:

Example 1.1 (Equivalent program in environment)

```
                          p = "bop(gt,uop(sqrt,fst(var("X"))), var("Y"))"
  |--- {1}
sqrt(x‘1) > y   ⟶   env = "LAMBDA (z : string) :
                     IF z = "X" THEN pairv(numv(x‘1),numv(x‘2))
                     ELSIF z = "Y" THEN numv(y) ELSE 0 ENDIF"
```

This string representation allows us to introduce these items in the current context with some PVS prover commands.

Equivalent `program` Computation. Given a PVS context and a statement `S`, by using the strategy language we can access to the corresponding lisp tree structure that represents the abstract syntax of the PVS statement. Therefore if the statement matches the embedded fragment, computing the equivalent `program` can be done by decomposing this lisp structure and building the corresponding string. As most of the cases are staightforward, we only detail a few of them:

- the variable: as mentioned earlier, the variables of the `program` type are identifiers (*e.g.,* string) and we need to have a mapping beetween every PVS variable and its corresponding string identifier.
- the projections: in PVS, tuples are represented as arrays (int → element), the corresponding lisp object is a list and we need to translate it as a binary tree, *e.g.,* list `(e1 e2 e3)` gives `pair(e1,pair(e2,e3))` and the projection `x‘3` is translated into `"snd(snd(Value"X"))"`

Corresponding Evaluation Environment. As we can see in example 1.1, the correspondence between identifiers and variables is not straightforward either. Indeed, we need to build the value corresponding to each identifier. Given an identifier `"X"` and its associated variable `x`, if `x` has a basic type, `number` or `bool`, then the semantics of `value("X")` is `x`, but if `x` is a tuple, then we need to extract its elements and build the corresponding `prog_val`. For example if `x` is a triple of type `[bool,real,real]` then the associated value `prog_val` is `pairv(boolv(x‘1),pairv(numv(x‘2),numv(x‘3)))`.

Given a PVS statement `E`, we are able to compute the corresponding `program` `p` and environment `env`, such that `E` can be replaced by the semantics of `p`,

i.e., `bo(sem(p,env))`. This allows us to work on the `program p` in order to apply the transformation.

2 Strategy Definition

In this section we present how to build the strategy that transforms a current goal or hypothesis into an equivalent square root and division free one. In the `program` expressions we will avoid writing constructors that are obvious, *e.g.,* we will write `"A"` and `plus(e1,e2)` instead of `val("A")` and `bop(plus,e1,e2)`.

Strategy Principles. Fig. 1 describes the main steps of the `elim-sqrt` strategy:

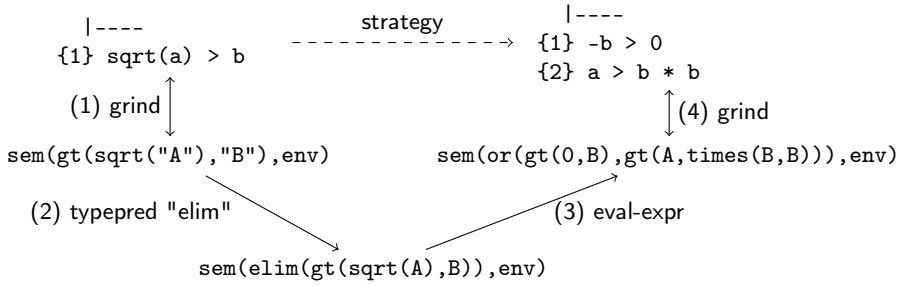


Fig. 1. `elim-sqrt` strategy outlines

(1) we introduce the equivalent `program` and environment and prove this equivalence using symbolic evaluation with grind
(2) using the type predicate of `elim` we apply this function to the `program`
(3) we compute the elimination using computational reflection eval-expr
(4) we return into the PVS language itself using symbolic evaluation of the square root and division free `program` semantics

In section 2, we gave the main steps of the transformation strategy, we will now see how these different expressions can be introduced in the PVS prover, and their equivalence proved. In this section we will assume that we have an hypothesis, `H`, we want to remove square roots from, the elimination in a positive formula (*e.g.,* a Goal) being similar.

From PVS Expression to `Program Datatype`. As mentioned in section 1 the transformation is defined using the `program` abstract datatype, the first step of the strategy is therefore to transpose the PVS statement into this datatype. In 1 we introduced a lisp function that, given a PVS statement, builds the corresponding `program, p` and environment `env`. The first step of the strategy is to introduce this `program` equivalent to `H` using its boolean semantics `bo(sem(p,env))`. The extraction of the boolean part of the semantics with `bo` such as the use of the type of the `elim` function will require to prove that `sem(p,env)` does not fail and is a boolean `prog_val`, this can be done by doing a symbolic evaluation

of `sem(p,env)` but this evaluation is not very efficient. Therefore in order to do it only once, we introduce explicitly this hypothesis with the following command:

```
(case "boolv?(sem(p,env)) AND bo(sem(p,env))")
```

This rule introduces a new hypothesis we first have to prove in the current context. The proof of `boolv?(sem(p,env)) AND bo(sem(p,env))` only uses the symbolic evaluation of `sem(p,env)` that produces `boolv(H)` and therefore finishes that case. Now that we have introduced `bo(sem(p,env))` equivalent to `H`, we can delete `H` from the context.

`elim` Function Introduction. We now want to eliminate square roots and divisions from `p`. Hence, we introduce the type of `elim(p)`, with the `typepred` command (1), `nofail(sem(p,env))` is straightforward using the `-2` hypothesis and thus it allows the use of the semantics equality to replace `p` by `elim(p)` (2):

(1)
```
{-1} nofail(sem(p,env)) IMPLIES
sem(p,env) = sem(elim(p),env)
{-2} boolv?(sem(p,env))
{-3} bo(sem(p,env))
{-4} Hypothesis
  |----
{1} Goal
```

(2)
```
{-1} boolv?(sem(elim(p),env))
{-2} bo(sem(elim(p),env))
{-3} Hypothesis
  |----
{1} Goal
```

Computational Reflection. The next step is to produce the equivalent square root and division free formula, this is done by computational reflection of `elim(p)`. The use of this technique requires two hypotheses:

- the function, (*i.e.,* `elim`) has to be completely defined with computable structures (*e.g.,* use list instead of sets), so there is a corresponding executable lisp function,
- the arguments have to be ground (do not contain any PVS variable), this is ensured by using identifiers to represent the original PVS variable, the link between these identifiers and variables being handled separately by `env`.

Therefore we can compute `elim(p)` in order to get the equivalent `program`, `p'`, free of square roots and divisions with the eval-expr strategy.

Semantics Evaluation. From our new square root and division free `program` `p'` we want to get the corresponding PVS expression. Therefore we have to compute the semantics of this `program`. This is done once again by symbolic evaluation and in the end we get a new PVS statement `H'`, equivalent to `H`, free of square roots and divisions. Square roots and divisions being eliminated in this hypothesis we can now continue the proof using our favorite arithmetic strategy.

Conclusion

We have described how to turn a PVS computable specification and the corresponding proof of a program transformation into a PVS strategy. We realized

it by doing a deep embedding of PVS inside PVS, using symbolic evaluation to prove the correspondence between PVS and its embedding when the transformation itself uses computational reflection. This kind of embedding can be generalized for any transformation defined in PVS on an abstract datatype representing a fragment of PVS.

This strategy has been tested on various examples, from simple comparisons to more complex statements that embed variable definitions and conditional expressions. The strategy takes between 20 sec to few minutes mainly depending on the number of square roots. These results can be explained by the low performances of the PVS symbolic evaluation whereas the transformation itself that uses reflection, is almost instantaneous.

This strategy is also the first step of a larger scale transformation that aims at eliminating square roots and divisions from full PVS specifications and producing a semantics equivalence proof certificate.

Acknowledgment. I would like to thank César Muñoz for the many ideas he had on this project. This work was supported by the Assurance of Flight Critical System's project of NASA's Aviation Safety Program at Langley Research Center under Research Cooperative Agreement No. NNL09AA00A awarded to the National Institute of Aerospace.

References

1. Archer, M., Vito, B.D., Muñoz, C.: Developing user strategies in PVS: A tutorial. In: Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics, STRATA 2003, NASA/CP-2003-212448, NASA LaRC, Hampton, VA, USA, pp. 23681–22199 (September 2003)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition: A synopsis. SIGSAM Bull. 10, 10–12 (1976)
4. Dutertre, B., de Moura, L.: 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)
5. Lescuyer, S., Conchon, S.: Improving coq propositional reasoning using a lazy cnf conversion scheme. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 287–303. Springer, Heidelberg (2009)
6. Neron, P.: A formal proof of square root and division elimination in embedded programs. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 256–272. Springer, Heidelberg (2012)
7. Owre, S., Rushby, J.M., Shankar, N.: Pvs: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
8. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)

The Picard Algorithm for Ordinary Differential Equations in Coq

Evgeny Makarov and Bas Spitters

Radboud University Nijmegen*

Abstract. Ordinary Differential Equations (ODEs) are ubiquitous in physical applications of mathematics. The Picard-Lindelöf theorem is the first fundamental theorem in the theory of ODEs. It allows one to solve differential equations numerically. We provide a constructive development of the Picard-Lindelöf theorem which includes a program together with sufficient conditions for its correctness. The proof/program is written in the Coq proof assistant and uses the implementation of efficient real numbers from the CoRN library and the MathClasses library. Our proof makes heavy use of operators and functionals, functions on spaces of functions. This is faithful to the usual mathematical description, but a novel level of abstraction for certified exact real computation.

Keywords: Coq, Exact real computation, Ordinary Differential Equations, Constructive mathematics, Type classes.

1 The Picard-Lindelöf Theorem

Before embarking on the formalization, we give a concise presentation of the mathematical ideas behind it. Let $v : [-a, a] \times [-K, K] \to \mathbf{R}$ be continuous such that $v(x, 0) = 0$. Assume that $L > 0$ is such that $aL < 1$ and

$$|v(x, y) - v(x, y')| \leq L|y - y'| \tag{1}$$

for all $x \in [-a, a]$ and $y, y' \in [-K, K]$. If v is differentiable in the second argument, then we can choose $L := \sup_x \sup_{y \in [-K,K]} \frac{d}{dy} v(x, y)$. Consider the initial value problem

$$f'(x) = v(x, f(x)), \quad f(0) = 0.$$

To solve this equation we define the Picard operator $Pf(t) := \int_0^t v(x, fx)dx$ and observe that a fixed point $f = Pf$ is a solution to the differential equation, which can be seen by differentiating both sides. To find such a fixed point, we first show that P is a contraction.

Lemma 1. *The Picard operator is a contraction, with constant $aL < 1$, on the metric space $C([-a, a], [-K, K])$. If $f \subset [-a, a] \times [-K, K]$, then so is Pf.*

* The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 463–468, 2013.

Proof.

$$\sup_{t\in[-a,a]}\left|\int_0^t v(x,fx)dx-\int_0^t v(x,gx)dx\right| \leq \sup_{t\in[-a,a]}\int_0^t |v(x,fx)-v(x,gx)|dx$$
$$\overset{(1)}{\leq} aL\|f-g\|_\infty$$

Here $\|h\|_\infty$ is $\sup_{x\in[-a,a]}|h(x)|$. Since $v(x,0)=0$,

$$|Pf(t)| = \left|\int_0^t v(x,fx)dx\right| \leq t \sup_{x\in[-a,a]}|v(x,fx)|$$
$$\leq a \sup_{x\in[-a,a]}|v(x,fx)-v(x,0)| \leq aLK \leq K.$$

□

We can now apply the Banach fixed point theorem to the Picard operator on the complete metric space $C([-a,a],[-K,K])$ and obtain a fixed point. Having completed this mathematical introduction, we now discuss its formalization in Coq.

2 A Computational Library for Analysis

We depend a huge code base, the CoRN library [1] combined with the recent MathClasses library [2,3]. Part of this work[1] is adapting code from the old library to the new coding style.

2.1 Metric Spaces Using Type Classes

As presented above, we want to apply the Banach fixed point theorem. So we first apply a type-class based presentation of metric spaces, roughly following [4,3]. This definition of metric spaces uses a closed ball relation `ball e x y` which intuitively means that $d(x,y)\leq e$. We define the completion monad on metric spaces and we define *complete* metric spaces as the existence of a retract of the embedding of X into its completion. The completion consists of regular functions, a refinement of the notion of a Cauchy sequence.

```
Class Limit := lim : RegularFunction → X.
Class CompleteMetricSpaceClass `{Limit} := cmspc :> Surjective reg_unit (inv := lim).
```

To be able to reuse some of the old results, we prove that each old complete metric space — using only records, but no type class automation — defines one based on type classes. We want to consider the complete metric space $C[-a,a]$, so we define closed metric subspaces determined by a ball.

We have defined various classes of functions — uniformly continuous, Lipschitz, and so on. In order to be able to treat them all at once, we define a type class. In this way we can define e.g. the supremum metric once for all relevant spaces of functions.

[1] `https://github.com/EvgenyMakarov/corn/tree/master/ode`

```
Class Func (T X Y : Type) := func : T → X → Y.
```

Here T is the type of the function space with func as coercion to the type of functions together with the declaration of Class Func. We need to be careful, since this introduces an equality on function spaces, determined by the metric space of functions, but we already have the extensional equality. We want Coq to automatically find the latter for us. Moreover, we want to prevent Coq from looping. This could happen in the following way. Suppose we define the equality on a metric space using the ball, then one way to find an equality is to find a ball relation for each e — informally a proof that two elements have zero distance. Consider:

```
Global Instance Linf_func_metric_space_ball : MetricSpaceBall T :=
  λ e f g, forall x, ball e (func f x) (func g x).
```

This has two class arguments: Func T X Y and MetricSpaceBall Y. If we put MetricSpaceBall Y first, then the equality on some type T may require a ball on a fresh Y (since Y is not in the conclusion of Linf_func_metric_space_ball), and this would call Linf_func_metric_space_ball again and require a ball on a fresh Y1, etc.

We can prevent this by using the fact that instances of the leftmost type class arguments are searched first. So we made Func T X Y the first Class argument in Linf_func_metric_space_ball. We have few instances of Func, and the first argument of Func in those instances is of the form, say, UniformlyContinuous X Y, or Lipschitz X Y. So, if T does not have this form, then the search for an instance of Func T X Y fails immediately. As a result, if T is, e.g., A → B, then the equality on T found by Coq is extensional equality, not the one is obtained not through MetricSpaceBall. For more on this kind of logic programming see [5,2].

The uniformly continuous functions between two complete metric spaces form a complete metric space. The distance between two functions may be infinite.

2.2 An Axiomatic Treatment of Integration

As explained in the first section, we turn a differential equation into an integral equation. To do so, we will now develop a theory of integration. There are at least two constructive Coq formalizations of the integral. The CoRN formalization closely follows Bishop's treatment of the Riemann integral. It computes in principle, but in practice it is impossible to evaluate it inside Coq. Hence it is not suitable for our purpose. As argued by Dieudonné, it seems better to treat the Cauchy integral (only continuous functions) and develop the full theory of Lebesgue integration when we need to go further. This is roughly the approach taken by Spitters and O'Connor [6] who developed Cauchy integration theory for $C[0,1]$ and then define the completion in order to obtain a flavor of Lebesgue integration. This version computes inside Coq. Here we develop a very similar approach based on type classes. This time, we choose a less abstract, but slightly faster treatment. We define the integral for locally uniformly continuous functions $\mathbf{Q} \to \mathbf{R}$ with an abstract specification similar to the one by Bridger [7, Ch5.].

```
Class Integral (f: Q → CR) := integrate: forall (from: Q) (w: QnonNeg), CR.
Class Integrable '{!Integral f}: Prop :=
  { integral_additive:
    forall (a: Q) b c, ∫f a b + ∫f (a+' b) c == ∫f a (b+c)%Qnn
  ; integral_bounded_prim: forall (from: Q) (width: Qpos) (mid: Q) (r: Qpos),
    (forall x, from ≤ x ≤ from+width → ball r (f x) ('mid)) →
    ball (width * r) ∫( f from width) (' (width * mid))
  ; integral_wd:> Proper (Qeq ⟹ QnonNeg.eq ⟹ @st_eq CRasCSetoid) ( ∫f) }.
```

Here CR is the completion of the rationals, i.e. the reals. The types Qpos and Qnn are the positive and nonnegative rational numbers respectively. The last line says that the integral respects the various setoid equalities. This specification is complete in that it uniquely characterizes the integral — that is, the Riemann sums approximate the integral to within arbitrary precision, so any two integrals must be setoid equal. The class thus expresses that a function $\int$ is an implementation of the integral. We provide a reference implementation using the technology of type classes. It is less abstract, but twice as fast as the development in [6].

2.3 Picard Iteration

As a final ingredient, we define the Picard operator from uniformly continuous functions to uniformly continuous functions. We define a function extend which extends a function from an interval to the real line in a constant way. This allows us to define:

```
Definition picard' (f : sx → sy) '{!IsUniformlyContinuous f mu} : Q → CR :=
  λ x, y0 + int (extend x0 rx (v ∘ (together Datatypes.id f) ∘ diag)) x0 x.
```

Here int is (basically) in instance of the type class Integral and (v ∘ (together Datatypes.id f)∘ diag) is a pointfree definition of $\lambda x.v(x, f(x))$. By defining this function using combinators we automatically define a uniformly continuous, and hence integrable, function. Moreover, the type class mechanism automatically proves that the extension

```
(extend x0 rx (v ∘ (together Datatypes.id f) ∘ diag))
```

from the ball with radius r_x around x_0 to the real line is integrable. This requires some care as all assumptions, such as 0 ≤ rx, need to be in the context. We then prove that the Picard operator is a contraction.

The Picard operator maps $[-a, a] \times [-K, K]$ to itself. Hence, we can iterate it and apply the Banach Fixed Point theorem.

```
Context '{MetricSpaceClass X}{Xlim : Limit X}{Xcms : CompleteMetricSpaceClass X}.
Context (f : X → X) '{!IsContraction f q} (x0 : X).
Let x n := nat_iter n f x0.
Let a := lim (reg_fun x _ cauchy_x).
Lemma banach_fixpoint : f a = a.
```

Applying this to the Picard operator on the metric space $C[-a, a]$, we find that there is an f such that $Pf = f$. This is the required solution to the integral equation.

2.4 Timings

We have made very initial experiments with computation inside Coq. For the differential equation $f = f'$, with solution $\lambda x, e^x$, we compute the value for $x = 1/2$. We use $x_0 = 0, a = 1/2, y_0 = 1, K = 1$ and $v(x, y) = y$ is uniformly continuous with modulus $\lambda e, e$ and Lipschitz on the second argument with $L = 1$.

Time results in seconds are in the table below. The rows correspond to requested approximation (10^{-n}) and the columns correspond to the number of iterations. An empty cell means that it took too long. Note that it takes 2 iterations (i.e., $1 + x + x^2/2$) to produce 1 correct digit after the decimal point and 3 iterations ($1 + x + x^2/2 + x^3/6$) to produce 2 correct digits. Evaluating the fixed point function takes too long, since the Banach fixpoint theorem requests more iterations than necessary.

	1	2	3	4
1	0	0	1	35
2	0	1	670	
3	0	216		

The timings show that our implementation performs reasonably well, but there are a number of possible improvements:

- Use reals based on dyadic rationals, as in [3].
- Use Newton iteration instead of Picard iteration. A variant of the work in [8] might be useful here.
- Use an improved algorithm for the integral such as Simpson integration. A constructive proof of Simpson integration can be found in [9].
- Use Coq's experimental native_compute; see [10].

Conclusion

We are working towards a verified implementation of a simple ODE-solver in Coq. We formally solved the integral equation. To show that this also solves the differential equation requires gluing the new code to the older formalization of the fundamental theorem of calculus. This should be straightforward, but has not been done yet.

We mention related work by Immler and Hölzl in Isabelle [11]. They go further in that they also implement the Euler method. At times, our implementation seems more natural, in that we can use dependent types to express for instance the type of continuous functions on a given interval. Their code can be extracted from Isabelle to SML and produces approximately two verified decimal digits. We obtain a bit less, but we compute inside the Coq proof assistant. Hence, all our computations are actually verified. Finally, we would like to mention the work on verifying a C-program for the wave equation [12,13].

Acknowledgements. We would like to thank Jelle Herold and Eelis van der Weegen who were involved in earlier initiatives to formalize the Picard Theorem.

References

1. 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)
2. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. MSCS, Special Issue on "Interactive Theorem Proving and the Formalization of Mathematics" 21, 1–31 (2011)
3. Krebbers, R., Spitters, B.: Type classes for efficient exact real arithmetic in Coq. LMCS 9(1:1) (2013), doi:10.2168/LMCS-9(1:01)2013
4. 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)
5. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In: ICFP, pp. 163–175 (2011)
6. O'Connor, R., Spitters, B.: A computer verified, monadic, functional implementation of the integral. TCS 411, 3386–3402 (2010)
7. Bridger, M.: Real analysis, a constructive approach. Pure and Applied Mathematics (New York). Wiley (2007)
8. Julien, N., Paşca, I.: Formal Verification of Exact Computations Using Newton's Method. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 408–423. Springer, Heidelberg (2009)
9. Coquand, T., Spitters, B.: A constructive proof of Simpson's rule. Logic and Analysis 4(15), 1–8 (2012)
10. Boespflug, M., Dénès, M., Grégoire, B.: Full reduction at full throttle. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 362–377. Springer, Heidelberg (2011)
11. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in Isabelle/HOL. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 377–392. Springer, Heidelberg (2012)
12. Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G., Weis, P.: Formal proof of a wave equation resolution scheme: the method error. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 147–162. Springer, Heidelberg (2010)
13. Boldo, S., Clément, F., Filliâtre, J., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. Journal of Automated Reasoning, 1–34 (2011)

Stateless Higher-Order Logic with Quantified Types

Evan Austin and Perry Alexander

The University of Kansas
Information and Telecommunication Technology Center
2335 Irving Hill Rd, Lawrence, KS 66045
{ecaustin,alex}@ittc.ku.edu

Abstract. There have been numerous extensions to classical higher-order logic, but not all of them interact non-trivially. Two such extensions, stateless HOL and HOL extended with quantified types, generate an interesting conflict in the way that type operator variables are implemented and handled. This paper details a proposed solution to that conflict and explores the key impacts to the logical kernel. A prototype system, implemented in GHC Haskell, is discussed and compared to related systems.

1 Introduction

LCF-style implementations of higher-order logic (HOL) have been in existence for a little over three decades. In that time, the original proof system has spawned an impressive family tree with each descendant imparting their own spin on the HOL design. This paper focuses on the confluence of the logics of two such systems: Freek Wiedijk's Stateless HOL [15] and Norbert Vöelker's HOL2P [14]. The intersection of these logics is problematic in that each introduces the notion of type operators in different ways to serve different purposes.

Taking a step back, it's important to address why these extensions are attractive to users in the first place. There's a growing desire in the Haskell community to formally verify software written in the language. Following from the "eat your own dog food" attitude, there's an equally strong inclination for those tools to be implemented in Haskell itself. There have been numerous attempts to meet these goals that we are aware of, but ultimately they all rely on passing the verification to an external tool at some point in the process [5,8]. It is our goal to simplify that work-flow by providing a general purpose theorem prover written in Haskell for Haskell.

We targeted HOL as the base logic for our proof system due to its applicability in a wide variety of problem domains [4,10,9] and its use in a large number of theorem provers with active communities [11,13,6]. After a less than successful attempt to naively translate HOL Light directly into Haskell [1], the authors sought out systems with modified logics that could assist their implementation

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 469–476, 2013.

efforts. Stateless HOL was attractive due to its reformulated logical kernel that represented a large leap towards a pure and total implementation that aligned more closely with Haskell's ideology. HOL2P was targeted for its potential to improve the connection between language and logic in the opposite direction, as it is based on a polymorphic lambda calculus that could be used to capture several of Haskell's language features more directly in HOL.

In this paper we make the following contributions. In Section 2 we review the primitive logics for the systems mentioned above, focusing on changes that are relative to our desired features. Then, in Section 3 we identify the inconsistency that occurs with the melding of these systems and propose a modified logic that solves the problem. Finally, in Section 4 we discuss a prototype system based on this logic, HaskHOL, that is implemented using GHC Haskell.

2 Background

The foundation of both Stateless HOL and HOL2P is John Harrison's HOL Light. The goal of HOL Light is to provide a full powered HOL proof assistant with a logical kernel that is simpler compared to those of related systems. The simplified kernel paired with an embedded domain-specific language implementation approach that allows HOL Light to inherit many of its primitive operations from its host language gives the entire system a very lightweight feel, as the name would imply.

Central to the logical kernel is the representation of HOL types and terms. HOL Light's elected representation maps almost directly to the simply-typed lambda calculus. The only significant difference between the two is that, rather than fixing the set of base types, HOL Light supports type extension through its `TyApp` constructor. The first field of the constructor is a string identifier for a type constant with the second field containing a list of type arguments to apply to the constant.

HOL Light includes two primitive type constants that represent boolean and function types. Any other type constants must be introduced through HOL Light's theory extension mechanisms. This restriction is necessary to guarantee that a user can not provide conflicting definitions of a constant within the same working theory. In order to facilitate this design decision, HOL Light leverages global memory references to both track constants as they are introduced and store any information associated with the constant.

2.1 Stateless HOL

Stateless HOL modifies the logical kernel of HOL Light in an effort to sever the dependence on global state for the previously mentioned primitive extension mechanisms. The principal idea behind the stateless modification is a simple one: embed properties of the kernel types directly such that you no longer need

to query global state in order to retrieve them. We focus on the embedding of type constant information given its relevance to our work:

```
data HOLType                          data TypeOp
  = TyVar String                        = TyPrim    String Int
  | TyApp TypeOp [HOLType]              | TyDefined String Int
                                            Theorem
```

Compared to the original HOL Light system, the first field in the `TyApp` constructor has been changed from `String` to a new auxiliary data type, `TypeOp`, representing type operators[1].

There are two possible cases for type operators: primitive operators and defined operators, those introduced via theory extension. In either case the `TypeOp` instance carries both the operator identifier and its arity; instances of defined type operators also carry their definitional theorem. This theorem can be used to differentiate between operators of the same name, where as previously global state was used to guarantee uniqueness of operator names.

2.2 HOL2P

Excluding the redefinition of the kernel types, the logic of Stateless HOL is largely unchanged compared to that of the original HOL Light system. This is not the case with HOL2P, as it looks to extend HOL Light's logic rather than reimplement it in an alternative manner. The "2P", in this case, refers to the move from a simply-typed lambda calculus to a second-order, polymorphic lambda calculus.

Added to the logic are universal types, type abstractions and combinations, and type operator variables, as made familiar by numerous System F based programming languages. Thanks to Coquand, however, we know that the combination of HOL and System F is inconsistent [3] To avoid this problem HOL2P introduces a "smallness" constraint on bound types: universal types cannot abstract over other universal types or type variables that are otherwise unconstrained.

HOL2P makes only one modification to the implementation of HOL types; the addition of a constructor for universal types. All other type features of the system are added via auxiliary definition or syntactic distinction in the parser. In the case of type operator variables, the only thing that separates them from regular type variables is the presence or lack of an `_` character prefixing the variable name. As will be discussed in the next section, we are of the opinion that the lack of a structural distinction among these elements complicates a number of primitive operations of the logic.

[1] Note that the data type implementation shown above is written in Haskell, not OCaml, using HaskHOL's data types to make comparisons between systems more direct.

3 Stateless HOL with Type Quantifiers

Excluding the additional rules that HOL2P added to bring congruence and beta reduction to the type level, both it and Stateless HOL share the same primitive rules of the original HOL Light system. Of these rules, only one had significant differences between its implementations:

$$\text{INST\_TYPE}\frac{[(ty_1, tv_1), ..., (ty_n, tv_n)] \qquad A \vdash t}{A[ty_1, ..., ty_n/tv_1, ..., tv_n] \vdash t[ty_1, ..., ty_n/tv_1, ..., tv_n]}$$

Implicit in the `INST_TYPE` rule is the definition of type instantiation; this is where the key change between systems occurs.

In the simplest case, type instantiation is a substitution performed over type variables. For Stateless HOL this substitution is trivially defined by the following rules:

$$\begin{array}{lll} x[ty/x] & = ty & \\ y[ty/x] & = y & (y \neq x) \\ (c\ a_1 ... a_n)[ty/x] & = (c\ a'_1 ... a'_n) & \end{array}$$

Note that we use a tick notation to indicate recursive application of substitution. The intent here is to show that in the `TyApp` case only the first field is left unchanged.

This is not the case in HOL2P where the first argument to a type application may be a type operator variable and, therefore, may be subject to substitution. To handle this possibility we need to add two additional rules. For the sake of completeness, we also show the name capture avoiding substitution rules for universal types:

$$\begin{array}{lll} (\_x\ a_1 ... a_n)[c/\_x] & = (c\ a'_1 ... a'_n) & (\text{arity } c = n) \\ (\_x\ a_1 ... a_n)[\Pi b_1 ..... \Pi b_m . ty/\_x] & = ty[a'_1/b_1 ... a'_n/b_m] & (m = n, ty \text{ is small}) \\ (\Pi x.a)[ty/x] & = (\Pi x.a) & \\ (\Pi y.a)[ty/x] & = (\Pi y.a') & (y \neq x, y \text{ is not free in } ty) \end{array}$$

In order to facilitate substitution of type operator variables, HOL2P dictates that they have two different, but conceptually equivalent, representations. The first representation is a string value that can be used as an argument to the `TyApp` constructor. The second representation is a type variable value that can be used as part of a substitution pair in the implementation of the above rules. In its `type_subst` function, HOL2P relies on the `mk_tyvar` and `dest_tyvar` methods to convert between these representations.

The type variable representation is nonsensical, though, as there are no constant terms that can inhabit types of that form. The only real purpose of this representation is to serve as a mechanism for making the inclusion of type operator variables in substitution environments well-typed. Furthermore, in the case where a type constant is to be substituted for a type operator variable,

the same trick must be played to represent the constant as a type variable. Given this reliance on disingenuous type representations, we find these rules to be ill-defined at best.

Regardless of our qualms, the integration of a stateless approach eliminates the alternative representation work around described above. In a stateless system, the `mk_type` function can no longer be used to query the current working theory for a unique type constant that matches a given name. This prevents type operators from being derived during substitution in the same manner as HOL2P, thus they must be constructed beforehand. This changes the type of a substitution pair involving type operator variables from `(HOLType, HOLType)` to `(?, TypeOp)`[2].

In the above explanation we indicate that the first type of the new substitution pair is unknown because it is dependent on a system's implementation of type operator variables. In HaskHOL, we elect to make the distinction of type operator variables purely structural by adding a new constructor to the `TypeOp` data type. As such, we end up with three separate substitution functions. The rules for these functions are shown below along with their associated argument types for clarity's sake:

(HOLType, HOLType) Substitution

$$
\begin{array}{lll}
x[ty/x] & = ty & \\
y[ty/x] & = y & (y \neq x) \\
(c\ a_1 ... a_n)[ty/x] & = (c\ a'_1 ... a'_n) & \\
(\Pi x.a)[ty/x] & = (\Pi x.a) & \\
(\Pi y.a)[ty/x] & = (\Pi y.a') & (y \neq x, y \text{ is not free in } ty)
\end{array}
$$

In all cases, ty must preserve the smallness of x.

(TypeOp, TypeOp) Substitution

$$
\begin{array}{ll}
x[c/\_x] & = x \\
(\_x\ a_1 ... a_n)[c/\_x] & = (c\ a'_1 ... a'_n)\ (\text{arity } c = n) \\
(\Pi x.a)[c/\_x] & = (\Pi x.a')
\end{array}
$$

(TypeOp, HOLType) Substitution

$$
\begin{array}{lll}
x[\Pi b_1 ..... \Pi b_m.ty/\_x] & = x & \\
(\_x\ a_1 ... a_n)[\Pi b_1 ..... \Pi b_m.ty/\_x] & = ty[a'_1/b_1 ... a'_n/b_m] & (m = n, ty \text{ is small}) \\
(\Pi y.a)[\Pi b_1 ..... \Pi b_m.ty/\_x] & = (\Pi y.a') & (y \text{ is not free in } ty)
\end{array}
$$

[2] Note that substitution in most HOL systems matches the mathematical definition, such that the first element of a pair is substituted for the second. In HaskHOL, we flip this ordering to more closely match other, popular, Haskell libraries. References to the types of substitution pairs in this paper match the HaskHOL ordering.

4 HaskHOL

As has been mentioned multiple times, HaskHOL is our prototype implementation of a stateless HOL system with quantified types. Going a step beyond Stateless HOL, we've striven to make the logical kernel of HaskHOL not only pure, but also total. The stateful layer of the system is built on top of this kernel using monads, as is the tradition when simulating side-effects in Haskell.

The implementation of the kernel types in HaskHOL follows closely from the union of Stateless HOL and HOL2P's primitive types. Only two significant changes have been made, both with the goal of being able to implement the substitution functions covered at the end of the previous section as near direct transcriptions of the rules that define them.

```
data HOLType
  = TyVarIn Bool String
  | TyAppIn TypeOp [HOLType]
  | UTypeIn HOLType HOLType
```

```
data TypeOp
  = TyOpVar String
  | TyPrim String Int
  | TyDefined String Int
      Theorem
```

The first, already discussed, change is the new constructor for type operator variables in the `TypeOp` data type. The second change is the embedding of smallness constraints in a new boolean field in type variables.

We elected to utilize Haskell's type class system to provide an ad hoc polymorphic view of the new substitution functions. This allows us to expose an interface to the user that is nearly identical in form and function to those of the HOL systems that have inspired HaskHOL's development. Using these polymorphic methods requires no additional work by the user with one small exception. Given Haskell's predilection for most general types, a type inference error will be thrown when a user attempts to use an empty list as a substitution environment. This is easily remedied by giving the list any suitable type annotation, as is done in this example: `mkConst "="([]::[(HOLType, HOLType)])`. Such cases occur infrequently in the system's implementation, so the overall inconvenience level is relatively low.

At the time of this writing, a release of the source code for the HaskHOL system is being made available on the first author's website at `http://people.eecs.ku.edu/~eaustin/`. A more complete version of the system which includes documentation should be available on Hackage in the near future.

5 Related Work

HaskHOL is not alone in trying to bring a general purpose theorem prover to the Haskell community. Agda [2], a combination of a dependently typed programming language and proof assistant, has been attempting to grow a sizable user base for the past few years. While not exactly fitting the "in Haskell for Haskell" paradigm that HaskHOL is striving to fill, Agda gets close thanks largely to the

fact that its syntax and design was heavily inspired by Haskell itself. Because of this, it is possible to translate between Haskell code and Agda specification with relative ease which seems to be one of its main appeals to the Haskell community.

MProver [12] is another proof system developed in Haskell. Like HaskHOL, it is a relative newcomer to the field and still appears to be in an early, developmental stage. The system shows great promise, though, especially with its approach to tackling verification of lazy code through equational reasoning. Rather than selecting an established logic as its base, the foundation of MProver appears to be a new, purpose built logic designed to target the more interesting aspects of the Haskell language.

On the HOL side of things, HaskHOL is hardly unique in its attempt to develop a different and improved proof system. HOL Omega [7] is one such system that we're keeping a close eye on. We're doing so largely in part because it is similarly influenced by HOL2P, though it chooses to go a different direction by making logical extensions to HOL4 rather than HOL Light. The unique portion of HOL Omega's logic that we're really interested in is the inclusion of a kind system similar to the one found in System Fω. As Haskell users, we're no strangers to using kinds in our day to day work, so we're excited to see if a proof system can capture that style of programming in a natural and straightforward way.

6 Conclusions and Future Work

The goal of HaskHOL has always been to be a general purpose theorem prover used in support of Haskell based projects. With the current iteration of its logic described in this paper we feel closer to achieving that goal than we ever have before. This is obviously largely thanks to the authors of the HOL systems that have inspired us up to this point. However, like any good research project, Haskell represents a moving target that seems to have an ever increasing velocity.

Recently, the work on kind promotion has had a serious impact on Haskell's type system and core language [16]. This change opens the door to a wide variety of useful programming techniques that HaskHOL will remain unable to verify until it adds a compatible kind system to its logic. HOL Omega was intentionally mentioned last in the related work section for this reason. It comes very close to what we desire for the next evolutionary step of HaskHOL. We're hoping to one day implement a similar kind system in order to close the capability gap between HaskHOL and Haskell as much as we can.

References

1. Austin, E., Alexander, P.: HaskHOL: A Haskell Hosted Domain Specific Language Representation of HOL Light. In: Trends in Functional Programming Draft Proceedings (2010)
2. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)

3. Coquand, T.: A New Paradox in Type Theory. In: Logic, Methodology and Philosophy of Science IX: Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science, pp. 7–14. Elsevier (1994)
4. Gordon, M.: Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, North-Holland (1986)
5. Haftmann, F.: From Higher-Order Logic to Haskell: There and Back Again. In: Gallagher, J.P., Voigtländer, J. (eds.) Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19. ACM (2010)
6. Harrison, J.: HOL Light: A Tutorial Introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
7. 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)
8. Huffman, B.: Formal Verification of Monad Transformers. In: Thiemann, P., Findler, R.B. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, September 9-15, pp. 15–16. ACM (2012)
9. Klein, G., Derrin, P., Elphinstone, K.: Experience Report: SEL4: Formally Verifying a High-Performance Microkernel. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, pp. 91–96. ACM, New York (2009)
10. Melham, T.: Higher Order Logic and Hardware Verification. Cambridge University Press, New York (1993)
11. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Procter, A., Harrison, W.L., Stump, A.: The Design of a Practical Proof Checker for a Lazy Functional Language. In: Trends in Functional Programming Draft Proceedings (2012)
13. Slind, K., Norrish, M.: A Brief Overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
14. Völker, N.: HOL2P - A System of Classical Higher Order Logic with Second Order Polymorphism. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 334–351. Springer, Heidelberg (2007)
15. Wiedijk, F.: Stateless HOL. In: Hirschowitz, T. (ed.) Proceedings Types for Proofs and Programs, Revised Selected Papers. EPTCS, vol. 53, pp. 47–61 (2009)
16. Yorgey, B.A., Weirich, S., Cretin, J., Jones, S.L.P., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a Promotion. In: Pierce, B.C. (ed.) Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, January 28, pp. 53–66. ACM (2012)

Implementing Hash-Consed Structures in Coq

Thomas Braibant[1], Jacques-Henri Jourdan[1], and David Monniaux[2,⋆]

[1] Inria
[2] CNRS / VERIMAG

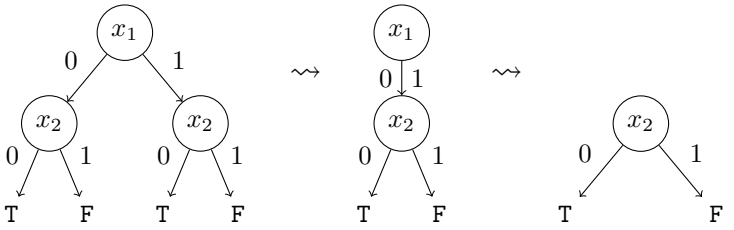
Abstract. We report on three different approaches to use hash-consing in programs certified with the Coq system, using binary decision diagrams (BDD) as running example. The use cases include execution inside Coq, or execution of the extracted OCaml code. There are different trade-offs between faithful use of pristine extracted code, and code that is fine-tuned to make use of OCaml programming constructs not available in Coq. We discuss the possible consequences in terms of performances and guarantees.

1 Introduction

Hash-consing is an implementation technique for immutable data structures that keeps a single copy, in a global hash table, of semantically equivalent objects, giving them unique identifiers and enabling constant time equality testing and efficient *memoization* (also known as *dynamic programming*). A prime example of the use of hash-consing is reduced ordered binary decision diagrams (ROBDDs, BDDs for short), representations of Boolean functions [3] often used in software and hardware formal verification tools, in particular *model checkers*.

A Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be represented as a complete binary tree with $2^n - 1$ decision nodes, labeled by variables x_i according to the depth from the root (thus the adjective *ordered*) and with subtrees labeled 0 and 1, and leaves labeled `T` (for true) or `F` (for false) . Such a tree can be *reduced* by merging identical subtrees, thus becoming a connected *directed acyclic graph* (see second diagram below); choice nodes with identical children are removed (see third diagram below). The reduced representation is *canonical*: a function is (up to variable ordering $x_1, \ldots, x_n$) represented by a unique ROBDD.

For instance, the function $f(0,0) = \mathtt{T}$, $f(0,1) = \mathtt{F}$, $f(1,0) = \mathtt{T}$, $f(1,1) = \mathtt{F}$ is represented, then simplified as:



⋆ This work was partially funded by ERC project "STATOR" and by Agence Nationale de la Recherche, grant number ANR-11-INSE-003 Verasco.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 477–483, 2013.

In practice, one directly constructs the reduced tree. To do so, a BDD library usually maintains a global pool of diagrams and never recreates a diagram that is isomorphic to one already in memory, instead reusing the one already present. In typical implementations, this pool is a global hash table. Hence, the phrase *hash consing* denotes the technique of replacing nodes creation by lookup in a hash table returning a preexisting object, or creation of the object followed by insertion into the table if previously nonexistent. A unique identifier is given to each object, allowing fast hashing and comparisons. This makes it possible to do efficient *memoization*: the results of an operation are tabulated so as to be returned immediately when an identical sub-problem is encountered. For instance, in a BDD library, memoization is crucial to implement the or/and/xor operations with time complexity in $O(|a|.|b|)$ where $|a|$ and $|b|$ are the sizes of the inputs; in contrast, the naive approach yields exponential complexity.

In this article, we investigate how hash-consing and memoization, imperative techniques, may be implemented using the Coq proof assistant, using the example of a BDD library, with two possible uses: 1) to be executed inside Coq with reasonable efficiency, e.g. for proofs by reflection; 2) or to be executed efficiently when extracted to OCaml, e.g. for use in a model-checking or static analysis tool proved correct in Coq.

2 A Problem and Three Solutions

In the following, we propose to implement a BDD library using three different approaches. We focus on a minimal set of operations: node creation, Boolean operations (or, and, xor, not) and equality testing on formulas represented as ROBDDs; and we provide formal guaranties of their correctness. (Note that, in some of our solutions, we do not prove the completeness of the equality test. That is, we prove that the equality test returning true implies equality of the formulas; but proving the converse is not essential for many applications.)

The typical way of implementing hash-consing (a global hash table) does not translate easily to Coq. The reason is that the Gallina programming language at the heart of the Coq proof assistant is a purely applicative language, without imperative traits such as hash tables, or pointers or pointers equality.

Therefore, there are two approaches to the implementation of hash-consing for data-structures in Coq. The first one is to model the memory using finite maps inside Coq, and use indices in the maps as surrogates for pointers, implementing all the aforementioned operations on these persistent maps. Such an implementation was described in [5,4], and we propose a new one in §2.1. The second one is to recover imperative features by fine-tuning the extraction of Coq code: either by realizing selected Coq constants by efficient OCaml definitions, e.g., extracting Coq constructors into smart OCaml constructors and fixpoint combinators into memoizing fixpoint combinators (see §2.2); or by explicitly declaring as axioms the OCaml code implementing the hash constructs and its properties (see §2.3).

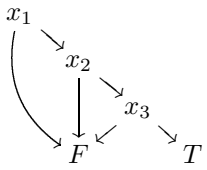
```
Inductive expr :=
  F | T | N : positive → expr.
Definition node := (expr * var * expr).
Record hashcons := {
graph: positive ⇝ node;
hmap : node ⇝ positive;
next : positive }.
```

```
Definition mk_node (l : expr) (v: var) (h : expr) st :=
if expr_eqb l h then (l,st)
else match find (l,v,h) (hmap st) with
       | Some x ⇒ (N x, st)
       | None ⇒ (N st.(next), upd (l,v,h) st)
     end.
```

Fig. 1. Hash-consing in pure Coq

2.1 Pure Coq

Our first implementation of BDDs is defined as follows in Coq. First, we assign a unique identifier to each decision node. Second, we represent the directed acyclic graph underlying a BDD as a Coq finite map from identifiers to decision nodes (that is, tuples that hold the left child, the node variable and the right child). For instance, the following graph, on the left, can be represented using the map on the right.



$$\begin{array}{lcl} 1 & \mapsto & (\mathrm{F},\, x_1,\, \mathrm{N}\ 2) \\ 2 & \mapsto & (\mathrm{F},\, x_2,\, \mathrm{N}\ 3) \\ 3 & \mapsto & (\mathrm{F},\, x_3,\, \mathrm{T}) \end{array}$$

Then, we implement the hash-consing pool using another map from decision nodes to node identifiers and a `next` counter that is used to assign a unique identifier to a fresh node. Equality between BDDs is then provided by decidable equality over node identifiers. We present on Fig. 1 our inductive definitions (left) and the code of the associated allocation function `mk_node` (right), knowing that `upd n st` allocates the fresh node `n` in the hash-consing state `st` (taking care of updating both finite maps and incrementing the "next fresh" counter).

We define well-formedness as follows. A node identifier is *valid* in a given global state when it is lower than the value of the `next` counter. Then, the notion of well-formedness of global states covers the facts that `graph` maps all valid node identifiers to valid nodes (nodes whose children are valid); and `hmap` is a left-inverse of `graph`.

Then, all operations thread the current global state in a monadic fashion that is, of course, reminiscent of a state monad. The correctness of BDD operations corresponds to the facts that 1) the global state is used in a monotonic fashion (that is the structure of the resulting global state is a refinement of the input one and that the denotation of expressions is preserved); 2) the resulting global state is well-formed; 3) the denotation of the resulting BDD expression is correct. As can be expected from our data structure, BDD operations cannot be defined using structural recursion (there is no inductive structure on which to recurse). Using well-founded recursion is difficult here because the well-founded relation involves both parameters of the function and the global state. Proving it to be well-founded would involve merging non-trivial proofs of monotonicity within programs. In the end, we resorted to define partial functions that use a *fuel* argument to ensure termination.

Finally, it is possible to enrich our hash-consing structure with memoization tables in order to tabulate the results of BDD operations.

```
Record memo := {
mand : (positive * positive) ⇝ expr;
mor  : (positive * positive) ⇝ expr;
mxor : (positive * positive) ⇝ expr;
mneg : positive ⇝ expr}.
```

```
Record BDD := { ...  :> hashcons; ...  :> memo}.
```

The memoization tables are passed around by the state monad, just as the hash-consing structure. It is then necessary to maintain invariants on the memoization information. Namely, we have to prove that the nodes referenced in the domain and in the codomain of those tables are valid; and that the memoization information is semantically correct.

As a final note: this implementation currently lacks garbage collection (allocated nodes are never destroyed until the allocation map becomes unreachable as a whole); it could be added e.g. by reference counting.

2.2 Smart constructors

In the previous approach, we use a state monad to store information about hash-consing and memoization. However, one can see that, even if these programming constructs use a mutable state, they behave transparently with respect to the pure Coq definitions. If we abandon efficient executability inside Coq, we can write the BDD library in Coq as if manipulating decision trees without sharing, then add the hash-consing and memoization code by tweaking the extraction mechanism. An additional benefit is that, since we use native hash tables, we may as well use *weak* ones, enabling the native garbage collector to reclaim unused nodes without being prevented from doing so by the pointer from the table.

More precisely, we define our BDDs as in Fig. 2a. Moreover, we tell Coq to extract the `bdd` inductive type to a custom `bdd` OCaml type (see left of Fig. 2b) and to extract constructors into smart constructors maintaining the maximum sharing property. These smart constructors make use of generic hash-consing library by Conchon and Filliâtre [2] that defines the α `hash_consed` type of hash-consed values of type α and the `hashcons` function that returns a unique hash-consed representative for the parameter. Internally, the library uses suitable hash and equality functions on BDDs together with weak hash tables to keep track of unique representatives.

In Coq, we define the obvious `bdd_eqb` function of type `bdd` $\rightarrow$ `bdd` $\rightarrow$ `bool`, that decides structural equality of BDDs. Then, we extract this function into OCaml's physical equality. From a meta-level perspective, the two are equivalent thanks to the physical unicity of hash-consed structures.

The last ingredient needed to transform a decision tree library into a BDD library is memoization. We implement it by using special well-founded fixpoint combinators in Coq definitions, which we extract into a memoizing fixpoint combinator in OCaml. As an example, we give the definition of the `bdd_not` operation in Fig. 2c. The fixpoint combinator is defined using the Coq general `Fix` well-founded fixpoint combinator that respects a fixpoint equality property. The definition of

```
Inductive bdd: Type :=
| T | F | N: var → bdd → bdd → bdd.
```

```
Extract Inductive bdd ⇒
  "bdd hash_consed" ["hT" "hF" "hN"] "bdd_match".
```

(a) BDDs in Coq as decision trees

```
type bdd =
| T | F
| N of var * bdd hash_consed * bdd hash_consed

let bdd_tbl = hashcons_create 257
```

```
let hT = hashcons bdd_tbl T
let hF = hashcons bdd_tbl F
let hN (p, b1, b2) = hashcons bdd_tbl (N(p, b1, b2))

let bdd_match fT fF fN b =
 match b.node with
 | T → fT () | F → fF ()
 | N(p, b1, b2) → fN p b1 b2
```

(b) Hash-consed OCaml BDD type

```
Definition memoFix1 :=
  Fix (well_founded_ltof bdd bdd_size).
Lemma memoFix1_eq : ∀ Ty F b,
  memoFix1 Ty F b =
  F b (fun b' _ ⇒ memoFix1 Ty F b').
Proof. [...]   Qed.
```

```
Program Definition bdd_not : bdd → bdd :=
  memoFix1 _ (fun b rec ⇒ match b with
                | T ⇒ F | F ⇒ T
                | N v bt bf ⇒
                  N v (rec bt _) (rec bf _)
              end).
```

(c) Using a fixpoint combinator for `bdd_not`

Fig. 2. Implementing BDDs in Coq, extracting them using smart constructors

```
Axiom var : Set.

Axiom uid : Set.
Axiom uid_eqb : uid → uid → bool.
Axiom uid_eq_correct: ∀ x y : uid,
    (uid_eqb x y =true) ↔ x=y.

Inductive bdd : Set :=
| T | F
| N : uid → var → bdd → bdd → bdd.

Axiom mkN : var → bdd → bdd → bdd.

Axiom mkN_ok :
∀ v : var, ∀ bt bf : bdd,
  ∃ id, mkN v bt bf = N id v bt bf.
```

```
Inductive valid : bdd → Prop :=
| valid_T : valid T
| valid_F : valid F
| valid_N : ∀ var bt bf,
  (valid bt) → (valid bf) →
  (valid (mkN var bt bf)).

Axiom shallow_equal_ok :
  ∀ id1 id2 : uid,
  ∀ var1 var2 : var,
  ∀ bt1 bf1 bt2 bf2 : bdd,
    valid (N id1 var1 bt1 bf1) →
    valid (N id2 var2 bt2 bf2) →
    id1 = id2 →
    N id1 var1 bt1 bf1 =
    N id2 var2 bt2 bf2.
```

Fig. 3. Axiomatization of equality using unique identifiers

`bdd_not` then uses `memoFix1` and requires proving that the BDDs sizes are decreasing (these trivial proof obligations are automatically discharged).

We extract the `memoFix1` combinator to a memoizing construct, that is observationally equivalent to the original one. However, this new construct tabulates results in order to avoid unnecessary recursive calls. We use similar techniques for binary operations. As all Coq definitions are kept simple, proofs are straightforward: we can prove semantic correctness of all operations directly using structural induction on decision trees.

2.3 Axioms

In the previous approach, hash-consing and memoization are done after the fact, and are completely transparent for the user. In the following, we make

more explicit the hypotheses that we make on the representation of BDDs. That is, we make visible in the inductive type of BDDs that each BDD node has a "unique identifier" field (see Fig. 3) and we take the node construction function as an axiom, which is implemented in OCaml. Note that nothing prevents the Coq program from creating new BDD nodes without calling this function `mkN`. Yet, only objects created by it (or copies thereof) satisfy the `valid` predicate; we must declare another axiom stating that unique identifier equality is equivalent to Coq's Leibniz equality *for valid nodes*. Then, we can use unique identifiers to check for equality.

This approach is close to the previous one. It has one advantage, the fact that unique identifiers are accessible from the Coq code. They can for instance be used for building maps from BDDs to other data, as needed in order to print BDDs as a linear sequence of definitions with back-references to shared nodes. Yet, one could also expose unique identifiers in the "smart constructor" approach by stating as axioms that there exists an injection from the BDD nodes to a totally ordered type of unique identifiers.

The use of axioms is debatable. On the one hand, the use of axioms somewhat lowers the confidence we can give in the proofs, and they make the code not executable within Coq. On the other hand, these axioms are actually used implicitly when extracting Coq constructors to "smart constructors": they correspond to the metatheoretical statement that these constructors behave as native Coq constructors. Thus, they make explicit some of the magic done during extraction.

3 Discussion

We compare our approaches on different aspects:

Executability Inside Coq. Both the "smart constructors" and the "pure" implementations can be executed inside Coq, even if the former has dreadful performances (when executed inside Coq, it uses binary decision trees). The "axiomatic" approach cannot be executed inside Coq.

Efficiency of the Extracted OCaml Code. We have yet to perform extensive testing, but preliminary benchmarks indicate that the "pure" approach yields code that is roughly five times slower than the "smart constructors" approach (and we assume that the latter is also representative of the "axiomatic" approach) on classic examples taken from previous BDD experiments in Coq [5]. We have yet to measure memory consumption.

Trust in the Extracted Code. Unsurprisingly, the "smart constructors" and the "axiomatic" approaches yield code that is harder to trust, while the "pure" approach leaves the extracted code pristine.

Proof. From a proof-effort perspective, the "smart constructors" is by far the simplest. The "axiomatic" approach involves the burden of dealing with axioms. However, it makes it easier to trust that what is formally proven corresponds to the real behavior of the underlying runtime. By comparison, the "pure" approach required considerably more proof-engineering in order to check the validity of invariants on the global state.

Garbage Collection. Implementing (and proving correct) garbage collection for the "pure" approach would require a substantial amount of work. By contrast, the "smart" and "axioms" approaches make it possible to use OCaml's garbage collector to reclaim unreachable nodes.

4 Conclusion and Directions for Future Works

In this paper, we proposed two solutions to implement hash-consing in programs certified with the Coq system. The first one is to implement it using Coq data-structures; the second is to use the imperative features provided by OCaml through the tuning of the extraction mechanism. The difference in flavor between the mapping of Coq constants to smart OCaml realizers or the axiomatization of there realizers in Coq is a matter of taste. In both cases, some meta-theoretical reasoning is required and requires to "sweep something under the rug".

We conclude with directions for future works. First, we believe that the smart constructors approach is generalizable to a huge variety of inductive types. One can imagine that it could be part of the job of Coq's extraction mechanism to implement on-demand such smart constructors and memoizers as it was the case for other imperative constructs [1]. Second, we look forward to investigate to what extent one could provide a certified version of the hash-consing library proposed by Conchon and Filliâtre [2].

Ackowledgements. We thank the reviewers for their helpful comments and Jean-Christophe Filliâtre for fruitful discussions.

References

1. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, Heidelberg (2010)
2. Conchon, S., Filliâtre, J.-C.: Type-safe modular hash-consing. In: ACM SIGPLAN Workshop on ML, Portland, Oregon (September 2006)
3. Knuth, D.E.: The Art of Computer Programming, Binary decision diagrams, vol. 4A, ch. 7.1.4. Addison-Wesley (2011)
4. Verma, K.N., Goubault-Larrecq, J.: Reflecting BDDs in Coq. Rapport de recherche RR-3859, INRIA (2000)
5. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting BDDs in Coq. In: Kleinberg, R.D., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 162–181. Springer, Heidelberg (2000)

Towards Certifying Network Calculus

Etienne Mabille[1], Marc Boyer[2], Loïc Fejoz[1], and Stephan Merz[3]

[1] RealTime at Work, Nancy, France
[2] ONERA, The French Aerospace Lab, Toulouse, France
[3] Inria & LORIA, Nancy, France

1 Result Certification for Network Calculus

Network Calculus (NC) [5] is an established theory for determining bounds on message delays and for dimensioning buffers in the design of networks for embedded systems. It is supported by academic and industrial tool sets and has been widely used, including for the design and certification of the Airbus A380 AFDX backbone [1,3,4]. However, while the theory of NC is generally well understood, results produced by existing tools have to be trusted: some algorithms require subtle reasoning in order to ensure their applicability, and implementation errors could result in faulty network design, with unpredictable consequences.

Tools used in design processes for application domains with strict regulatory requirements are subject to a qualification process in order to gain confidence in the soundness of their results. Nevertheless, given the safety-critical nature of network designs, we believe that more formal evidence for their correctness should be given. We report here on work in progress towards using the interactive proof assistant Isabelle/HOL [6] for certifying the results of NC computations. In a nutshell (cf. Figure 1), the NC tool outputs a trace of the calculations it performs, as well as their results. The validity of the trace (w.r.t. the applicability of the computation steps and the numerical correctness of the result) is then established offline by a trusted checker.

The approach of result certification is useful in general for computations performed at design time, as is the case with the use of NC tools, and the idea of using interactive theorem provers for result certification is certainly not new. In particular, it is usually easier to instrument an existing tool in order to produce a checkable trace than to attempt a full-fledged correctness proof. Also, the NC tool can be implemented by a tool provider using any software development process, programming language, and hardware, and it can be updated without having to be requalified, as long as it still produces certifiable traces.

In the remainder, we give a brief introduction to NC, outline our ongoing work on formalizing NC in Isabelle/HOL, and finally illustrate its use for the certification of bounds on the message delay in a toy network.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 484–489, 2013.

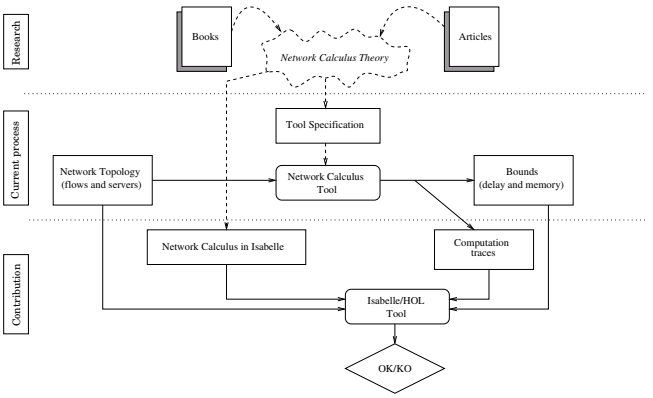


Fig. 1. Proof by instance of NC computations

2 Network Calculus

Network calculus [5] is a theory for computing upper bounds in networks. Its mathematical background is a theory of the set of functions

$$\mathcal{F} = \{\, f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\} \mid x \leq y \implies f(x) \leq f(y) \,\} \tag{1}$$

that form a dioid under the operations $\sqcap$ and $+$ defined as pointwise minimum and addition. Practical applications make frequent use of four families of functions, defined as $\delta_d(t) = 0$ if $t \leq d$ and ∞ otherwise, $\beta_{R,T}(t) = 0$ if $t \leq T$ and $R(t-T)$ otherwise, and $\gamma_{r,b}(t) = 0$ if $t \leq 0$ and $rt + b$ otherwise (all parameters denote real numbers).

Operations of interest on $\mathcal{F}$ include convolution $*$, deconvolution $\oslash$, and the sub-additive closure f^*.

$$(f * g)(t) = \inf_{0 \leq u \leq t} (f(t-u) + g(u)) \tag{2}$$

$$(f \oslash g)(t) = \sup_{0 \leq u} (f(t+u) - g(u)) \tag{3}$$

$$f^* = \delta_0 \sqcap f \sqcap (f * f) \sqcap (f * f * f) \sqcap \cdots \tag{4}$$

A *flow* is represented by its cumulative function $R \in \mathcal{F}$, where $R(t)$ is the total number of bits sent by this flow up to time t. A flow R has function $\alpha \in \mathcal{F}$ as *arrival curve* (denoted $R \preceq \alpha$) if $\forall t, s \geq 0 : R(t+s) - R(t) \leq \alpha(s)$, meaning that, from any instant t, the flow R will produce at most $\alpha(s)$ new bits of data in s time units. Using convolution, this condition can be equivalently expressed as $R \leq R * \alpha$. If α is an arrival curve for R, so is α^*, and also any $\alpha' \geq \alpha$.

$$R \preceq \alpha \implies R \preceq \alpha^* \tag{5}$$

$$R \preceq \alpha, \alpha \leq \alpha' \implies R \preceq \alpha' \tag{6}$$

A *server* S is a relation between an input flow R, and an output flow R' (denoted $R \xrightarrow{S} R'$) such that $R' \leq R$ (representing the intuition that the flow crosses the

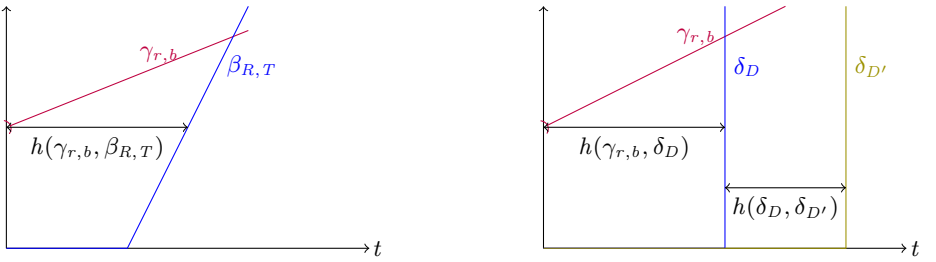


Fig. 2. Common curves and delay

server, and that the output is produced after the input). Such a server has a *service curve* β if $R' \geq R * \beta$ holds. The delay incurred by the flow R can be bounded by the maximal horizontal difference between curves α and β, formally defined as

$$h(\alpha, \beta) \;=\; \sup_{s \geq 0} \left(\inf \{\, \tau \geq 0 \mid \alpha(s) \leq \beta(s+\tau) \,\}\right) \tag{7}$$

(cf. also Fig. 2). If R has arrival curve α and S has service curve β then $\alpha \oslash \beta$ and $\alpha \oslash \delta_{h(\alpha,\beta)}$ are two possible arrival curves for R'.

This presentation gives a flavor of network calculus as a collection of algebraic results useful for computing bounds on curves and delays. Consider a configuration with a flow R crossing two servers S_1, S_2 in sequence: $R \xrightarrow{S_1} R' \xrightarrow{S_2} R''$. Assume that R has arrival curve α and that each server S_i offers a service of curve β_i. Then, the delay of R in S_1 can be bounded by $d = h(\alpha, \beta_1)$, and $\alpha' = \alpha \oslash \delta_d$ is a possible arrival curve for R'. Its sub-additive closure $(\alpha')^*$ is also an arrival curve for R' (by Eq. 5), but may be too expensive to compute. A simpler approximation is given by $\alpha' \sqcap \delta_0 \geq (\alpha')^*$ (using Eq. 6), and the delay of R' in S_2 can be bounded by $h(\alpha' \sqcap \delta_0, \beta_2)$. The end-to-end delay can also be bounded by the sum of bounds on local delays, *i.e.* $h(\alpha, \beta_1) + h(\alpha' \sqcap \delta_0, \beta_2)$.

This simple example illustrates that implementations of NC analysis may choose between different approximations, involving tradeoffs between the accuracy of the result, the difficulty of implementing the necessary computations, and their time complexity.

3 Encoding Network Calculus in Isabelle

The first step towards developing a result certifier consists in formalizing the theory underlying NC to the extent that it is used by algorithms we are interested in. As a side benefit of this formalization, we obtain a rigorous development of NC, including all possible corner cases that may be overlooked in pencil-and-paper proofs. The objective of the work reported here was to evaluate the feasibility of developing a result certifier in Isabelle that would at least be able to check computations for simple, but representative networks. Our NC formalization is

currently incomplete, with many theorems only partly proved; we nevertheless outline the main definitions and results.

The set $\mathcal{F}$ (Eq. 1) of non-decreasing functions used to represent flows is represented in Isabelle/HOL as the type

typedef $ndf = \{ f :: ereal \Rightarrow ereal \;.\; (\forall r \leq 0.\, f\, r = 0) \wedge mono\; f \}$

where *ereal* is a pre-defined type corresponding to $\mathbb{R} \cup \{\infty\}$. Compared to (1), we extend the domain of f to $\mathbb{R} \cup \{\infty\}$ (including negative numbers and ∞) but require that $f\, r$ be zero for negative r. This insignificant change of definition turned out to simplify the subsequent development. Over type *ndf*, we define operations such as addition, multiplication, and comparison by pointwise extension and establish some basic algebraic properties: for example, the resulting structure forms an ordered commutative monoid with 0 and 1.

We introduce operations such as convolution and deconvolution (Eq. 2) and characteristic properties such as sub-additivity, and prove fundamental results. For example, the convolution of two sub-additive flows is itself sub-additive.[1]

definition *is-sub-additive* **where**
is-sub-additive $f \;\equiv\; \forall x\, y.\, f{\cdot}(x + y) \leq f{\cdot}x + f{\cdot}y$
lemma *convol-sub-add-stable*:
assumes *is-sub-additive* f **and** *is-sub-additive* g
shows *is-sub-additive* $(f * g)$

A *simple server* is represented as a left-total relation between flows such that the output flow is not larger than the input flow

typedef $server = \{\, s :: (ndf \times ndf)\; set.\; (\forall in.\, \exists out.\, (in, out) \in s) \wedge (\forall (in, out) \in s.\; out \leq in) \,\}$

and we define what it means for a flow to be constrained by an arrival curve α and for a server to provide minimum service β:

$$R \preceq \alpha \;\equiv\; R \leq R * \alpha \qquad S \trianglerighteq \beta \;\equiv\; \forall (in, out) \in S : in * \beta \leq out.$$

Again, we prove results relating these constraints to bounds on delays and backlogs. For example, the following theorem provides a bound on the delay of a simple server:

theorem *d-h-bound*:
assumes $in \preceq \alpha$ **and** $S \trianglerighteq \beta$
shows *worst-delay-server in* $S \leq$ *h-dev* $\alpha\ \beta$

where the horizontal deviation is defined in (Eq. 7) and *worst-delay-server in S* denotes the maximal delay incurred by input flow *in* at server S.

Building on these results about simple servers, we derive theorems about sequences of servers. We also formalize concepts such as *packetization*, which refers to servers that group individual bits into larger packets, introducing extra delays. Finally, these concepts are extended to multiple-input multiple-output servers that takes vectors of flows as input and output. We do not describe these concepts in detail, as they are not used in the following example.

[1] $f{\cdot}x$ denotes the result of applying $f :: ndf$ to x.

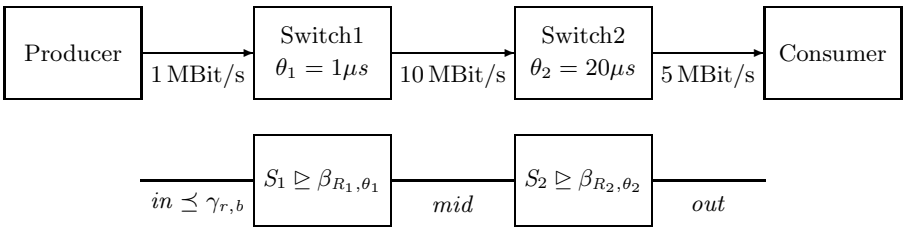


Fig. 3. A simple system and its Network Calculus representation.

4 Certifying a Simple Network Computation

In order to illustrate the use of our theories on a simple example, let us consider the producer-consumer setup shown in Fig. 3. The producer is assumed to send at most one frame every $T = 20\,\text{ms}$. We further assume that the maximum frame size is $MFS = 8000\,\text{bits}$. This flow is sent to a consumer via two switches with switching delays $\theta_1 = 1\,\mu s$ and $\theta_2 = 20\,\mu s$. The physical links between the producer, the switches, and the consumer are assumed to have bandwidths of 1, 10 and 5 MBit/s, respectively.

The NC model appears in the lower part of Fig. 3. Flow *in* is constrained by the arrival curve $\alpha_{in} = \gamma_{r,b}$ where b equals MFS and $r = \frac{MFS}{T} = \frac{8000}{20\times10^3} = \frac{2}{5}$.

The service curves are given by the function β_{R_i,θ_i} where the bandwidths are $R_1 = 10\,\text{bit}/\mu s$ and $R_2 = 5\,\text{bit}/\mu s$, and the delays are θ_1 and θ_2.

We are interested in the maximal delays that frames may incur. Using theorem *d-h-bound*, the delay at server 1 is bounded by $h(\alpha_{in}, \beta_{10,1})$, which evaluates to $801\,\mu\text{s}$. As explained in Section 2, the arrival curve of flow *mid* can be computed as

$$\alpha_{mid} = (\alpha_{in} \oslash \delta_{801}) \sqcap \delta_0 = (\gamma_{\frac{2}{5},8000} \oslash \delta_{801}) \sqcap \delta_0 = \gamma_{\frac{2}{5},\frac{41602}{5}}.$$

Continuing for the second server, its delay is at most $h(\alpha_{mid}, \beta_{5,20}) = \frac{42102}{25}\,\mu\text{s}$. Consequently, the overall delay incurred by frames equals

$$801\,\mu\text{s} + \frac{42102}{25}\,\mu\text{s} = \frac{62127}{25}\,\mu\text{s}.$$

These computations are performed by the PEGASE Network Calculus tool [2] and certified using Isabelle.

5 Conclusion

We have presented preliminary work aiming at ensuring the correctness of embedded network designs by certifying the result of standard NC tools within a theory developed in the proof assistant Isabelle/HOL. A prototype has been developed and it can handle a realistic industrial configuration, with 8 switches and more than 5.000 flows, in 8 hours on a standard laptop computer. Much remains to be

done: the proofs of many theorems of the NC formalization are still incomplete. Moreover, we only support simple arrival curves and therefore obtain worse bounds than state-of-the-art tools for NC analysis. Nevertheless, we believe that our work demonstrates the feasibility and the interest of the approach.

Developing a Network Calculus engine that is able to handle an AFDX configuration requires about one or two years of implementation. The effort for developing a qualified version of such an engine, using state-of-the-art techniques (documentations, testing, peer-review, etc.) is higher by a factor of 5 or 10.

Although one should not confuse result certification with the development of a qualified NC tool, the approach that we suggest here promises to reduce the overhead while increasing the confidence in the results produced by the software. We have so far invested less than 1 development year for encoding some fundamental concepts of Network Calculus in Isabelle/HOL, and for instrumenting an existing tool so that it produces a trace that can be checked in Isabelle. We estimate that the overall effort for producing the proof for a realistic network should be between 2 and 3 years. This includes effort to complete the formalization of the basic concepts, extensions to more complicated types of servers, and developing special-purpose proof methods for checking the proof traces.

In other words, we believe that result certification could reduce the overhead for developing a trustworthy version of a Network Calculus tool to a factor of 2 or 3, while significantly improving its quality.

References

1. AEEC. Arinc 664p7-1 aircraft data network, part 7, avionics full-duplex switched ethernet network. Technical report, Airlines Electronic Engineering Committee (September 2009)
2. Boyer, M., Navet, N., Olive, X., Thierry, E.: The PEGASE project: Precise and scalable temporal analysis for aerospace communication systems with network calculus. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, pp. 122–136. Springer, Heidelberg (2010), `http://www.realtimeatwork.com/software/rtaw-pegase/`
3. Frances, F., Fraboul, C., Grieu, J.: Using network calculus to optimize AFDX network. In: Proc. 3thd Europ. Cong. Embedded Real Time Software (ERTS 2006), Toulouse (January 2006)
4. Grieu, J.: Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques. PhD thesis, Institut National Polytechnique de Toulouse (INPT), Toulouse (June 2004)
5. Le Boudec, J.-Y., Thiran, P.: Network Calculus. LNCS, vol. 2050. Springer, Heidelberg (2001)
6. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

Steps towards Verified Implementations of HOL Light

Magnus O. Myreen[1], Scott Owens[2], and Ramana Kumar[1]

[1] Computer Laboratory, University of Cambridge, UK
[2] School of Computing, University of Kent, UK

Abstract. This short paper describes our plans and progress towards construction of verified ML implementations of HOL Light: the first formally proved soundness result for an LCF-style prover. Building on Harrison's formalisation of the HOL Light logic and our previous work on proof-producing synthesis of ML, we have produced verified implementations of each of HOL Light's kernel functions. What remains is extending Harrison's soundness proof and proving that ML's module system provides the required abstraction for soundness of the kernel to relate to the entire theorem prover. The proofs described in this paper involve the HOL Light and HOL4 theorem provers and the OpenTheory toolchain.

1 Introduction

We are developing a new verification friendly dialect of ML, called CakeML. This ML dialect is approximately a subset of Standard ML carefully carved out to be convenient to program in and to reason about formally. We plan to build verified implementations of CakeML (a compiler, an implementation of a read-eval-print loop and possibly custom hardware) and also produce tools for generating and reasoning about CakeML programs (e.g. tools for synthesising CakeML from high-level specifications). One of our initial challenge examples is to construct and verify implementations of HOL Light [1] expressed in CakeML.

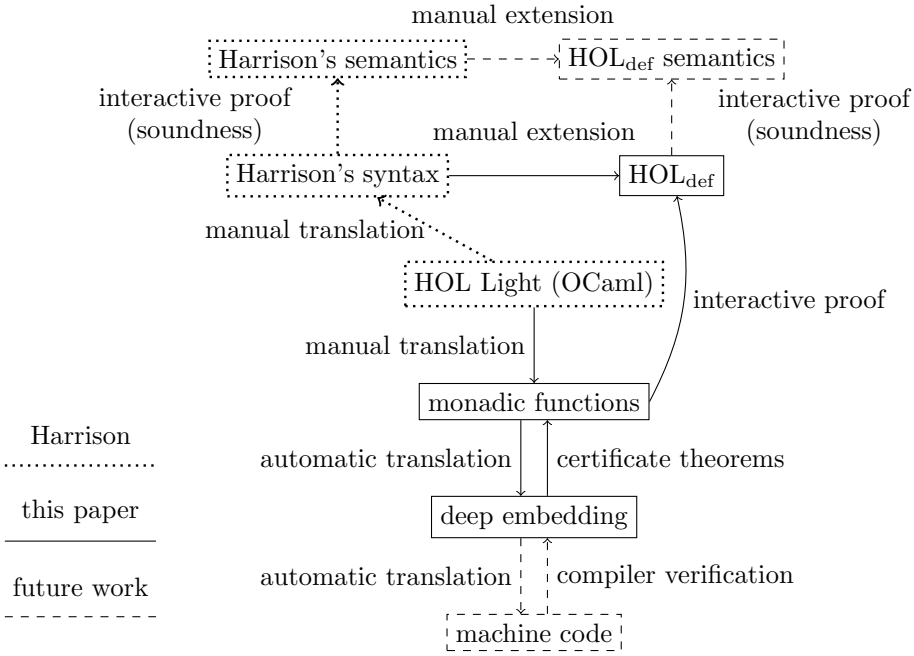
This short paper describes our plans and progress towards our HOL Light case study, which we believe could be the first formal proof of soundness for an implementation of an LCF-style prover. The theorem we are aiming for relates the semantics of higher-order logic (i.e. HOL) with the execution of the concrete machine code which runs the prover. We want to prove that only sound theorems can be derived in our CakeML implementations.

We build on previous work where Harrison [2] has formalised HOL in the HOL Light prover, and on our proof-producing synthesis tool [5] for MiniML, a pure version of CakeML. Throughout, we will simply write ML for CakeML.

2 Method

The following are the high-level steps we have taken in our effort to construct verified implementations of HOL Light.

S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.): ITP 2013, LNCS 7998, pp. 490–495, 2013.



1. **Extending Harrison's formalisation of HOL.** We started by adding support for user-defined constants, types and axioms to Harrison's specification of the syntactic inference rules of HOL. (We have yet to update Harrison's soundness proof, i.e. this extension of the inference rules has yet to be proved sound.) We will refer to this extension as HOL_{def}.

2. **Kernel as monadic functions.** Next, we took the sources for the HOL Light kernel (`fusion.ml` in the HOL Light code repository) and manually translated each HOL Light kernel function (written in OCaml) into a definition inside HOL. Since the OCaml code is stateful and uses exceptions, these HOL functions were written using a state-exception monad. We will refer to these functions as *the monadic functions*.

3. **Verification of the monadic functions.** We then proved that any computation the monadic functions can perform is something HOL_{def} allows: if the monadic functions allow construction of a theorem *thm* then *thm* is also derivable in HOL_{def}. If HOL_{def} is proved sound, then the monadic functions are also sound.

   Note that, Harrison's formalisation lives within the HOL Light theorem prover, but the rest of our work lives within the HOL4 theorem prover [7]. To bridge this gap, we transported our extension of Harrison's development from HOL Light into HOL4 using the OpenTheory toolchain [3]. OpenTheory replays the primitive inferences from one prover inside another.

4. **Verified kernel in ML.** Next, we constructed an actual ML implementation, a *deep embedding*, from the monadic functions. We constructed this

ML implementation using an *automatic* shallow-to-deep-embedding translator [5] which, for each translation, proves a certificate theorem w.r.t. a formal specification of the operational semantics of ML. These certificate theorems allowed us to carry over the correctness results for each monadic function (shallow embedding) over to the ML code (the deep embedding).

Our original translator only produced pure ML functions. For this work, we extended our previously developed translator to map the state-exception monads to appropriate stateful ML constructs.

The final — and currently missing — steps lift our kernel verification to a soundness result for the entire theorem prover running on a verified ML runtime:

5. **Verified theorem prover in ML.** We then hope to package up the deep embedding constructed above into an ML module (the HOL Light kernel) and prove that ML's module system provides the necessary restrictions which imply that only the kernel can construct values of type theorem (`thm`).
6. **Verified theorem prover running on a verified ML runtime.**

The rest of this paper describes these steps and discusses related work.

2.1 Formalising HOL with a Definition Mechanism

As mentioned above, we build on Harrison's formalisation of HOL inside HOL. We extended his model of the syntax with support for user-defined type operators (`Tyapp`) and term constants (`Const`):

```
type = Tyvar string | Bool | Ind | Fun type type
     | Tyapp string (type list)                    We added this line

term = Var string type | Equal type | Select type
     | Comb term term | Abs string type term
     | Const string type                           ... and this line.
```

We also define the 'state' of the logic. The state consists of a list of definitions: a definition defines a new constant, type or axiom:

```
def = Constdef string term                  term name, expression
    | Typedef string term string string     type name, prop, abs, rep
    | Axiomdef term                         statement of axiom
```

The inference rules were extended to include the new state component. Each judgement `hyps` $\vdash$ `concl` is now `defs, hyps` $\vdash$ `concl`, where `defs` is a list of definitions, i.e. `defs` has HOL type `def list`. There are also five new inference rules: one which allows extension of the definitions with a new definition,

$$\texttt{defs, asl} \vdash \texttt{p} \wedge \texttt{def\_ok d defs} \implies \texttt{(CONS d defs), asl} \vdash \texttt{p}$$

and four inference rules which provide theorems that arise from the definitions. For example, the following inference rule provides a description of a term definition. The constant `name` is equal to term `tm`, if the constant is defined as such in the list of definitions `defs` (which must be well-formed).

```
context_ok defs ∧ MEM (Constdef name tm) defs
⟹ defs, [] ⊢ Const name (typeof tm) === tm
```

Every attempt was made to be as minimal as possible in the extension of Harrison's work. The hope is that his semantics and soundness proof can be updated to work with this extension of his original formalisation. No attempt has yet been made to extend his semantics or soundness proof.

2.2 Defining the HOL Light Kernel in HOL Using Monads

We use implementation friendly versions of the main datatypes when defining the kernel of HOL Light as functions in HOL.

```
hol_type = Tyvar string | Tyapp string (hol_type list)

hol_term = Var string hol_type | Const string hol_type
         | Comb hol_term hol_term | Abs hol_term hol_term

thm = Sequent (hol_term list) hol_term
```

The kernel of HOL Light makes use of exceptions and maintains state. The state consists of three references: `the_type_constants`, `the_term_constants`, `the_axioms`. When defining the kernel of HOL Light as functions in HOL (the monadic functions), we model the state using a record. This record contains two new components: `the_definitions` keeps track of the 'state' of the logic; and `the_clash_var` is used to hold data that should be carried in an exception.[1]

```
hol_refs = <| the_type_constants : (string # num) list ;
              the_term_constants : (string # hol_type) list ;
              the_axioms         : thm list ;
              the_definitions    : def list ;
              the_clash_var      : hol_term |>
```

We then defined each function of HOL Light's kernel using a state-exception monad based on this record type. We make use of HOL4's special syntax for monads (due to Michael Norrish). For example, HOL Light's `mk_const` function

```
let mk_const(name,theta) =
  let uty = try get_const_type name with Failure _ ->
    failwith "mk_const: not a constant name" in
  Const(name,type_subst theta uty)
```

is defined in HOL as follows:

```
mk_const(name,theta) =
  do uty <- try get_const_type name
       "mk_const: not a constant name" ;
     return (Const name (type_subst theta uty))
  od
```

[1] At the time of writing, CakeML did not support carrying of arbitrary information in exceptions. This use of an extra reference is our temporary workaround.

The monad-bind operator that hides under the syntactic sugar propagates the state and exceptions appropriately. In some cases, the monadic version is necessarily more verbose than the original OCaml code, e.g.

```
let REFL tm = Sequent([],mk_eq(tm,tm))
```

must be split with a semicolon since `mk_eq` is a monadic function:

```
REFL tm = do eq <- mk_eq(tm,tm); return (Sequent [] eq) od
```

For each of these functions (`mk_const`, `REFL`, etc.) we proved that the types, terms, theorems and states they produce are wellformed, given wellformed inputs. A theorem is wellformed if it is derivable ($\vdash$) in our extension of Harrison's formalisation of HOL w.r.t. the current list of definitions (`the_definitions`).

2.3 Proof-Producing Synthesis of Stateful ML

The HOL Light kernel, as defined above, carries around state. In the generated ML, we implement this state using five references, one for each component of the state record from above. In order to use our previously developed proof-producing synthesis tool [5], we had to extend it with support for making use of such top-level references.

The extension essentially just threads a state (from the monadic functions) and reference store (from the ML semantics) through the entire development. At each point, the state and the reference store must agree according to a refinement invariant which relates the two representations of state.

The new state-aware synthesis tool produces deep embeddings and certificate theorems much like the original tool. For example, the monadic function `REFL` from above turns into the following ML code. Bind is translated into ML `let`.

```
val REFL = fun tm =>
  let val eq = mk_eq (Pair tm tm) in Sequent ([], eq) end;
```

The automatically proved certificate theorem for `REFL` makes a statement about the generated ML code (deep embedding) w.r.t. the operational semantics of ML: if the kernel has been loaded, then the name `"REFL"` refers to an ML function (deep embedding), which given an input, returns an output and accesses the state in a manner that exactly follows the monadic function (shallow embedding).

The details of this extension of our synthesis tool will be described in a forthcoming extension of the original conference publication [5].

3 Results, Discussion and Related Work

At the time of writing, we have a verified ML implementation (deep embedding) for each function in HOL Light's kernel. We have proved that all types, terms and theorems this ML code produces are wellformed w.r.t. our extension of Harrison's formalisation of HOL. What remains is: proving HOL_{def} sound; proving that the

module system successfully prevents construction of theorems (values of type `thm`) outside of the kernel; and construction of verified implementations of ML.

Why not verify HOL light as it is? Such a proof would require dealing with a semantics of OCaml [6]. Real OCaml includes problematic features such as mutable strings and unsafe primitives (e.g. `Obj.magic`), which can be used to seemingly or actually produce unsoundness in HOL Light. As mentioned at the beginning, our interest lies in developing a verification friendly ML dialect.

Would Wiedijk's stateless version of HOL light [8] have been easier to verify? Wiedijk's version of HOL is very neat. However, the fact that Harrison's HOL Light is stateful is not a major hurdle and Harrison's work on formalising HOL inside HOL fits better with his version of HOL Light. Our initial efforts concentrate on Harrison's stateful version, but we are also looking into constructing verified implementations of Wiedijk's stateless version.

What is the most closely related project? Our previous project [4] on proving soundness of Davis' ACL2-inspired Milawa theorem prover had similar aims: to prove that every theorem admitted by the Milawa system (when run on our implementation of Lisp) must be true by the semantics of the Milawa logic.

Acknowledgements. Freek Wiedijk initially got us started by asking: "Can you do for HOL Light what you did for Milawa?" We are also grateful for encouragement from John Harrison and appreciate comments received from Mike Gordon and Dan Synek on drafts this paper. The first author was funded by the Royal Society, UK, and the third author was funded by the Gates Cambridge Trust, UK.

References

1. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009), `http://www.cl.cam.ac.uk/~jrh13/hol-light/`
2. Harrison, J.: Towards self-verification of HOL Light. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 177–191. Springer, Heidelberg (2006)
3. Hurd, J.: The OpenTheory standard theory library. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 177–191. Springer, Heidelberg (2011)
4. Myreen, M.O., Davis, J.: The reflective Milawa theorem prover is sound (2012), `http://www.cl.cam.ac.uk/~mom22/jitawa/`
5. Myreen, M.O., Owens, S.: Proof-producing synthesis of ML from higher-order logic. In: Thiemann, P., Findler, R.B. (eds.) International Conference on Functional Programming (ICFP). ACM (2012)
6. Owens, S.: A sound semantics for OCaml light. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008)
7. 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)
8. Wiedijk, F.: Stateless HOL. In: Hirschowitz, T. (ed.) Types for Proofs and Programs (TYPES). EPTCS (2009)

Author Index

Alexander, Perry 469
Anand, Abhishek 261
Aravantinos, Vincent 295
Armstrong, Alasdair 197
Asperti, Andrea 163
Austin, Evan 469
Avigad, Jeremy 163

Bertot, Yves 163
Bickford, Mark 261
Blanchette, Jasmin Christian 35
Bodin, Martin 354
Bolignano, Dominique 1
Boyer, Marc 484
Braibant, Thomas 477

Chaudhuri, Kaustuv 386
Claret, Guillaume 67
Cock, David 311
Cohen, Cyril 163, 213

Endrullis, Jörg 354

Fejoz, Loïc 484

Garillot, François 163
Geuvers, Herman 451
Gonthier, Georges 163
González Huesca, Lourdes del Carmen 67

Haftmann, Florian 100
Hasan, Osman 295
Hendriks, Dimitri 354
Heule, Marijn J.H. 229
Hölzl, Johannes 279
Huffman, Brian 133, 279
Hunt Jr., Warren A. 229, 435

Immler, Fabian 279

Jourdan, Jacques-Henri 477

Kaliszyk, Cezary 35, 51, 451
Kaufmann, Matt 435
Krauss, Alexander 51, 100
Kühlwein, Daniel 35
Kumar, Ramana 490
Kunčar, Ondřej 100

Lammich, Peter 84
Leino, K. Rustan M. 2
Le Roux, Stéphane 163
Liu, Liya 295
Lochbihler, Andreas 116

Mabille, Etienne 484
Mahboubi, Assia 19, 163
Makarov, Evgeny 463
Manolios, Panagiotis 18
Merz, Stephan 484
Miyamoto, Kenji 370
Monin, Jean-François 338
Monniaux, David 477
Myreen, Magnus O. 490

Neron, Pierre 457
Nipkow, Tobias 100
Nordvall Forsberg, Fredrik 370
Norrish, Michael 133

O'Connor, Russell 163
Ould Biha, Sidi 163
Owens, Scott 490

Pasca, Ioana 163
Polonowski, Emmanuel 402
Pous, Damien 180

Rager, David L. 435
Rahli, Vincent 261
Régis-Gianas, Yann 67
Rideau, Laurence 163

Schürmann, Carsten 17
Schwichtenberg, Helmut 370
Shi, Xiaomu 338
Solovyev, Alexey 163
Spitters, Bas 463
Struth, Georg 197

Tahar, Sofiène 295
Tankink, Carst 451
Tassi, Enrico 19, 163
Théry, Laurent 163
Thiemann, René 245

Urban, Christian 147
Urban, Josef 35, 451

Vafeiadis, Viktor 328

Weber, Tjark 197
Wenzel, Makarius 418
Wetzler, Nathan 229

Xu, Jian 147

Zhang, Xingyuan 147
Ziliani, Beta 67