Lennart Beringer
Amy Felty (Eds.)

LNCS 7406

Interactive Theorem Proving

Third International Conference, ITP 2012
Princeton, NJ, USA, August 2012
Proceedings

Springer

Lennart Beringer Amy Felty (Eds.)

Interactive Theorem Proving

Third International Conference, ITP 2012
Princeton, NJ, USA, August 13-15, 2012
Proceedings

Springer

Volume Editors

Lennart Beringer
Princeton University
Department of Computer Science
35 Olden Street
Princeton, NJ 08540, USA
E-mail: eberinge@cs.princeton.edu

Amy Felty
University of Ottawa, School of Electrical Engineering and Computer Science
800 King Edward Ave.
Ottawa, ON K1N 6N5, Canada
E-mail: afelty@eecs.uottawa.ca

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-32346-1 e-ISBN 978-3-642-32347-8
DOI 10.1007/978-3-642-32347-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: Applied for

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 2012, the Third International Conference on Interactive Theorem Proving. The conference was held August 13–15 in Princeton, New Jersey, USA, organized by the General Co-chairs Andrew W. Appel and Lennart Beringer.

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 2012 was the third annual conference in this series. The first meeting was held July 11–14, 2010, in Edinburgh, UK, as part of the Federated Logic Conference (FLoC). The second meeting took place August 22–25, 2011, in Berg en Dal, The Netherlands. ITP evolved from the previous TPHOLs series (Theorem Proving in Higher-Order Logics), which took place every year from 1988 to 2009.

There were 40 submissions to ITP 2012, each of which was reviewed by at least four Program Committee members. Out of the 40 submissions, 36 were regular papers and four were rough diamonds. Unlike previous editions of TPHOLs/ITP, this year's 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 were thoroughly inspected by the reviewers and influenced the acceptance decisions. The Program Committee accepted 25 papers, which include 21 regular papers and four 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: Gilles Barthe (IMDEA, Spain), Lawrence Paulson (University of Cambridge, UK), and André Platzer (Carnegie Mellon University, USA). In addition, the Program Committee invited Andrew Gacek (Rockwell Collins) to give a tutorial on the Abella system. We thank all these speakers for also contributing articles to these proceedings.

ITP 2012 also featured two associated workshops held the day before the conference: The Coq Workshop 2012 and Isabelle Users Workshop 2012, bringing together users and developers in each of these communities to discuss issues specific to these two widely used tools.

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 2012. 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 referees enlisted by Program Committee members. Finally, we would like to thank Andrew W. Appel and his staff for taking care of all the local arrangements, Princeton University for the administrative and financial support, and NEC Laboratories, Princeton, for their additional sponsorship.

June 2012

Lennart Beringer
Amy Felty

Conference Organization

General Co-chairs

Andrew Appel	Princeton University, USA
Lennart Beringer	Princeton University, USA

Program Co-chairs

Lennart Beringer	Princeton University, USA
Amy Felty	University of Ottawa, Canada

Program Committee

Andreas Abel	LMU Munich, Germany
Nick Benton	Microsoft Research Cambridge, UK
Stefan Berghofer	secunet Security Networks AG, Germany
Lennart Beringer	Princeton University, USA
Yves Bertot	INRIA Sophia-Antipolis, France
Adam Chlipala	MIT, USA
Ewen Denney	SGT/NASA Ames, USA
Peter Dybjer	Chalmers University of Technology, Sweden
Amy Felty	University of Ottawa, Canada
Herman Geuvers	Radboud University of Nijmegen, The Netherlands
Georges Gonthier	Microsoft Research Cambridge, UK
Jim Grundy	Intel Corp., USA
Elsa Gunter	University of Illinois at Urbana-Champaign, USA
Hugo Herbelin	INRIA Roquencourt-Paris, France
Joe Hurd	Galois, Inc., USA
Reiner Hähnle	Technical University of Darmstadt, Germany
Matt Kaufmann	University of Texas at Austin, USA
Gerwin Klein	NICTA/University of New South Wales, Australia
Assia Mahboubi	INRIA Saclay, France
Conor McBride	University of Strathclyde, UK
Alberto Momigliano	University of Milan, Italy
Magnus O. Myreen	University of Cambridge, UK
Tobias Nipkow	TU Munich, Germany
Sam Owre	SRI, USA

Christine Paulin-Mohring	Université Paris-Sud, France
David Pichardie	INRIA Rennes, France
Brigitte Pientka	McGill University, Canada
Randy Pollack	Harvard University, USA
Julien Schmaltz	Open University of the Netherlands
Bas Spitters	Radboud University of Nijmegen, The Netherlands
Sofiene Tahar	Concordia University, Canada
Makarius Wenzel	Université Paris-Sud, France

Additional Reviewers

Abbasi, Naeem
Andronick, June
Appel, Andrew W.
Aravantinos, Vincent
Boespflug, Mathieu
Boldo, Sylvie
Brown, Chad
Bubel, Richard
Cave, Andrew
Chamarthi, Harsh Raju
Contejean, Evelyne
Dockins, Robert
Dominguez, Cesar
Filliâtre, Jean-Christophe
Gustafsson, Daniel
Hölzl, Johannes
Jacobs, Bart
Ji, Ran
Joosten, Sebastiaan
Kennedy, Andrew
Khan-Afshar, Sanaz
Krebbers, Robbert
Krstić, Sava
Licata, Daniel R.
Liu, Liya
Makarov, Evgeny
Matthews, John
McKinna, James
Melquiond, Guillaume
Mhamdi, Tarek
Murray, Toby
O'Connor, Russell
Paganelli, Gabriele
Payet, Etienne
Popescu, Andrei
Pous, Damien
Preoteasa, Viorel
Román-Díez, Guillermo
Schmidt, Renate
Senjak, Christoph-Simon
Sewell, Thomas
Siles, Vincent
Spiwack, Arnaud
Stewart, Gordon
Swierstra, Wouter
Trinder, Phil
Tuttle, Mark
Urban, Christian
van Gastel, Bernard
Verbeek, Freek
Wehrman, Ian
Wetzler, Nathan
Zeilberger, Noam

Table of Contents

MetiTarski: Past and Future

Lawrence C. Paulson

Computer Laboratory, University of Cambridge, England
lp15@cl.cam.ac.uk

Abstract. A brief overview is presented of MetiTarski [4], an automatic theorem prover for real-valued special functions: ln, exp, sin, cos, etc. MetiTarski operates through a unique interaction between decision procedures and resolution theorem proving. Its history is briefly outlined, along with current projects. A simple collision avoidance example is presented.

1 Introduction

MetiTarski [4] is an automatic theorem prover for first-order logic over the real numbers, including the transcendental and other special functions. It is a version of Joe Hurd's Metis [23,24] (a resolution theorem prover) heavily modified to call decision procedures, and more recently, augmented with case-splitting by backtracking.

Here are some theorems that MetiTarski can prove, automatically of course, and typically in tens of seconds.

$$\begin{aligned}&\forall\, t > 0, v > 0\\ &\quad ((1.565 + 0.313\, v)\cos(1.16\, t) + (.0134 + .00268\, v)\sin(1.16\, t))\, e^{-1.34\, t}\\ &\qquad\qquad - (6.55 + 1.31\, v)\, e^{-0.318\, t} + v \;\geq\; -10\end{aligned}$$

$$\forall\, x > 0 \Longrightarrow \frac{1 - e^{-2\,x}}{2\,x\,(1 - e^{-x})^2} - \frac{1}{x^2} \leq \frac{1}{12}$$

$$\forall\, x\, y,\; x \in (0, 12) \Longrightarrow xy \leq \frac{1}{5} + x\,\ln(x) + e^{y-1}$$

$$\forall\, x \in (-8, 5) \Longrightarrow \max(\sin(x), \sin(x+4), \cos(x)) > 0$$

$$\forall x\, y,\; (0 < x < y \wedge y^2 < 6) \Longrightarrow \frac{\sin(y)}{\sin(x)} \leq 10^{-4} + \frac{y - \frac{1}{6}y^3 + \frac{1}{120}y^5}{x - \frac{1}{6}x^3 + \frac{1}{120}x^5}$$

$$\forall x \in (0, 1) \Longrightarrow 1.914\frac{\sqrt{1+x} - \sqrt{1-x}}{4 + \sqrt{1+x} + \sqrt{1-x}} \leq 0.01 + \frac{x}{2 + \sqrt{1 - x^2}}$$

$$\forall x \in (0, 1.25) \Longrightarrow \tan(x)^2 \leq 1.75\, 10^{-7} + \tan(1)\tan(x^2)$$

$$\forall x \in (-1, 1), y \in (-1, 1) \Longrightarrow \cos(x)^2 - \cos(y)^2 \leq -\sin(x+y)\sin(x-y) + 0.25$$

$$\forall x \in (-1, 1), y \in (-1, 1) \Longrightarrow \cos(x)^2 - \cos(y)^2 \geq -\sin(x+y)\sin(x-y) - 0.25$$

$$\forall x \in (-\pi, \pi) \Longrightarrow 2\,|\sin(x)| + |\sin(2\,x)| \leq \frac{9}{\pi}$$

The motivation for MetiTarski arose with Jeremy Avigad's Isabelle/HOL proof of the prime number theorem. Avigad [7, Sect. 4.5] observed that some quite elementary inequalities involving the logarithm function (the proof requires establishing many such inequalities) were inordinately difficult to prove. My original idea was to create a simple-minded heuristic procedure within Isabelle to prove inequalities involving continuous functions. It might reason using monotonicity, backwards chaining, and ideas similar to those in the Fourier-Motzkin [26] approach to deciding real inequalities. Although decision procedures are popular, reasoning about arbitrary special functions is obviously undecidable. The decidable problems that do exist are very complex, both in theory and in practice. Our approach has to be heuristic.

2 Early Work

The initial work was done (using funding from the UK's EPSRC) by my colleague, Behzad Akbarpour. He located a promising paper [29] (see also Daumas et al. [16]) giving formally verified upper and lower bounds (polynomials or rational functions, namely, ratios of polynomials) for the well-known transcendental functions and describing techniques using interval arithmetic to establish ground inequalities involving special functions over the real numbers. Hand simulations quickly established that interval arithmetic was seldom effective at solving problems even in one variable [5].

Equipped with the upper and lower bounds, we could replace the transcendental functions appearing in a problem by polynomials. We could therefore reduce the original special-function inequality to a set of polynomial inequalities. Because interval arithmetic was too weak to decide these inequalities, we decided to try a decision procedure. First-order formulas over polynomial inequalities over the real numbers admit quantifier elimination [20], and are therefore decidable. This decision problem is known as RCF, for real closed fields.

The first implementation of MetiTarski [2] used John Harrison's implementation [27] of the Cohen-Hörmander RCF decision procedure. It turned out to be much more effective than interval arithmetic. As the procedure was itself coded in ML, MetiTarski was a self-contained ML program. Unfortunately, we found [2] that the Cohen-Hörmander procedure couldn't cope with polynomials of degree larger than five, which ruled out the use of accurate bounds. The next version of MetiTarski [3] invoked an external decision procedure, QEPCAD [12], which implements a much more powerful technique: cylindrical algebraic decomposition (CAD). More recently, we have integrated MetiTarski with the computer algebra system Mathematica, which contains a family of highly advanced RCF decision procedures. We can even use the SMT solver Z3 [18] now that it has been extended with nlsat, a novel approach to deciding purely existential RCF problems [25].

A separate question concerned how to apply the upper and lower bounds to eliminate special function occurrences. Should we write a bespoke theorem prover implementing a carefully designed algorithm? Analytica [14] and Weierstrass [8] both implement a form of sequent calculus. Despite the impressive results obtained by both of these systems, we decided to see whether ordinary first-order resolution could perform as well. The contest between "natural" and resolution proof systems is a long-standing scientific issue in automated reasoning [10]. Resolution had two strong arguments in its favour: one, its great superiority (in terms of performance) over any naive implementation of first-order logic, and two, the possibility that resolution might also be superior to a bespoke algorithm at finding complicated chains of reasoning involving the upper and lower bounds.

Resolution has entirely met our expectations. No individual bound can accurately approximate a special function over the entire real line. This was already clear in Muñoz and Lester [29], who presented families of bounds, each accurate over a small interval. A typical proof involves choosing multiple bounds over overlapping intervals of the real line. With resolution, we can supply bounds as files of axioms. Resolution automatically identifies bounds that are accurate enough, keeping track of the intervals for which the theorem has been proved. Moreover, other facts (such as the definitions of the functions abs and max) can be asserted declaratively as axioms.

The original families of bounds also underwent many refinements. Through careful scrutiny, we were able to extend their ranges of applicability. A later version of MetiTarski [4] adopted many continued fraction approximations [15]. For some functions, notably ln and $\tan^{-1}$, continued fractions are vastly more accurate than truncated Taylor series. There is also the choice between inaccurate but simple bounds (linear ones are helpful if functions are nested), and highly accurate bounds of high degree. Although the most recent version of MetiTarski can choose axiom files automatically, manually choosing which bounds to include can make a difference between success and failure for some problems. To extend MetiTarski to handle a new function, the most important step is to find approximations to the function that yield suitable upper and lower bounds.

3 Basic Architecture

MetiTarski comprises a modified resolution theorem prover, one or more RCF decision procedures and a collection of axiom files giving approximations (upper and lower bounds) to special functions. A few other axioms are used, relating division with multiplication, defining the absolute value function, etc.

The most important modifications to the resolution method are arithmetic simplification and the interface to RCF decision procedures. Polynomials are simplified in an ad hoc manner, loosely based on Horner canonical form; this serves first to identify obviously equivalent polynomials, but second and crucially to identify a special function occurrence to be eliminated. The mathematical formula is transformed to move this candidate occurrence into a certain position, which will allow an ordinary resolution step to replace it by an upper or lower bound, as appropriate. Occurrences of the division operator are also simplified and in particular flattened, so that an algebraic expression contains no more than a single occurrence of division, and that outermost.

RCF decision procedures are mainly used to simplify clauses by deleting certain literals. Recall that a clause is a disjunction of literals, each of which is an atomic formula or its negation. With standard resolution, a literal can only be deleted from a clause after a resolution step with another clause containing that literal's negation. MetiTarski provides another way a literal can be deleted: whenever an RCF decision procedure finds it to be inconsistent. This determination is done with respect to the literal's context, consisting of the negation of other literals in the same clause, as well as any globally known algebraic facts.

RCF decision procedures are also used to discard redundant clauses. Whenever a new clause emerges from the resolution process, it is given to the decision procedure, and if the disjunction of its algebraic literals turns out to be a logical consequence (in RCF) of known facts, then this clause is simply ignored. This step prevents the buildup of redundant algebraic facts, which cannot influence future decision procedure calls.

Our use of RCF decision procedures has one massive disadvantage: performance. The general decision problem is doubly exponential in the number of variables [17]. We actually use a special case of the decision problem, in which we ask whether universally quantified formulas are theorems, but even so, when a proof takes a long time, almost invariably this time is spent in decision procedures. The time spent in resolution is frequently just a few seconds, a tiny fraction of the total.

We undertook this project as basic research, motivated by Avigad's difficulties but with no other specific applications. Having no problem set to begin with, we (mainly Akbarpour) created our own. The original problems mostly came from mathematical reference works [1,13,28]. Later, we formalised engineering problems, including Nichols plot problems from Ruth Hardy's thesis [21], and simple hybrid systems problems that were published on a website [35]. The hardware verification group at Concordia University supplied a few more problems. We now have nearly 900.

4 Current Projects

Recent developments fall under three categories: improvements to the resolution process, the use of new decision procedures, and applications.

Resolution can be tweaked in countless ways, but one of the most important heuristics is splitting. This involves taking a clause of the form $A \vee B$ (all non-trivial clauses are disjunctions) and considering the cases A and B as separate problems. Given that resolution already can deal with disjunctions, splitting is not always appropriate, and indeed splitting is only possible if A and B have no variables in common. We have experimented [11] with two kinds of splitting: lightweight (essentially a simulation, implemented by adding propositional variables as labels in clauses) and with backtracking (similar to the splitting done by SAT-solvers). Backtracking is complicated to implement in the context of resolution theorem proving: it affects many core data structures, which must now be saved and restored according to complicated criteria; this work was done by James Bridge. Both forms of splitting deliver big performance improvements. The problem is only split into separate cases when each involves a special function problem.

The decision procedure is a crucial element of MetiTarski. From the first experiments using interval arithmetic and the first implementation, using the Cohen-Hörmander method, each improvement to the decision procedure has yielded dramatic improvements overall. We have the most experience with QEPCAD, but in the past year we have also used Mathematica and Z3. Each has advantages. QEPCAD is open source and delivers fast performance on problems involving one or two variables. Unfortunately, QEPCAD is almost unusable for problems in more than three variables, where Mathematica and Z3 excel. Grant Passmore is heavily involved with this work. With Mathematica, we can solve problems in five variables, and with Z3, we have sometimes gone as high as nine. (See the example in Sect. 6.) But we do not have the luxury of regarding the decision procedure as a black box. That is only possible when the performance bottlenecks are elsewhere. We have to examine the sort of problems that MetiTarski produces for the decision procedures, and configure them accordingly. Such tuning delivers significant improvements [30].

5 Applications

We have assumed that, as MetiTarski's performance and scope improved, users would find their own applications. Meanwhile, we have ourselves investigated applications connected with hybrid systems, and in particular with stability properties. Note that MetiTarski is not itself a hybrid system verifier: it knows nothing about the discrete states, transitions and other elements of a hybrid system. The discrete aspect of a hybrid system must be analysed by other means, but MetiTarski can assist in verifying the continuous aspect [6]. There have also been a few simple experiments involving the verification of analogue circuits [19].

The ability to prove theorems combining first-order logic and real-valued special functions is unique to MetiTarski. Mathematica can establish such properties in simple cases, as can certain constraint solvers [34], but these tools do not deliver proofs. MetiTarski delivers resolution proofs, which specify every detail of the reasoning apart from the arithmetic simplification and the decision procedure calls. Even these reasoning steps do not necessarily have to be trusted, as discussed below (Sect. 7).

Mathematicians will find that MetiTarski proofs are seldom natural or elegant. Mathematical properties of a special function are typically proved from first principles, referring to the function's definition and appealing to general theorems. A MetiTarski proof is typically a complicated case analysis involving various bounds of functions and signs of divisors. Compared with a mathematician's proof, such a proof will frequently yield a less general result (perhaps limited to a narrow interval). For these reasons, MetiTarski is more appropriate for establishing inequalities that arise in engineering applications, inequalities that hold for no simple reason.

6 A Collision Avoidance Problem

A problem studied by Platzer [32, Sect. 3.4] concerns collision avoidance for two aircraft, named x and y. For simplicity, we consider only two dimensions: the aircraft are flying in the XY plane.

The coordinates of aircraft x are written (x_1, x_2). Here, the subscript 1 refers to the X component of the aircraft's position, while subscript 2 refers to the Y component. (I am not to blame for this confusing notation!) Similarly, the velocity vector in two dimensions is written (d_1, d_2).

Glossing over the details of the derivation, here is a summary of the system of equations governing this aircraft:

$$\begin{array}{llll} x_1'(t) = d_1(t) & x_2'(t) = d_2(t) & d_1'(t) = -\omega d_2(t) & d_2'(t) = \omega d_1(t) \\ x_1(0) = x_{1,0} & x_2(0) = x_{2,0} & d_1(0) = d_{1,0} & d_2(0) = d_{2,0} \end{array}$$

This system admits a closed-form solution, yielding the trajectory of aircraft x:

$$\begin{aligned} x_1(t) &= x_{1,0} + \frac{d_{2,0}\cos(\omega t) + d_{1,0}\sin(\omega t) - d_{2,0}}{\omega} \\ x_2(t) &= x_{2,0} - \frac{d_{1,0}\cos(\omega t) - d_{2,0}\sin(\omega t) - d_{1,0}}{\omega} \end{aligned}$$

The treatment of aircraft y, whose coordinates are written (y_1, y_2), is analogous. We would like to prove that two aircraft following the trajectory equations, for certain ranges of initial locations and linear velocities will maintain a safe distance (called p, for "protected zone"):

$$(x_1(t) - y_1(t))^2 + (x_2(t) - y_2(t))^2 > p^2$$

Figure 1 presents the corresponding MetiTarski input file. MetiTarski can prove the theorem, but the processor time is 924 seconds.[1] Of this, six seconds are devoted to resolution proof search and the rest of the time is spent in the RCF decision procedure (in this case, Z3). The problem is difficult for MetiTarski because of its nine variables. It is only feasible because of our recent research [30] into heuristics such as model sharing (which can eliminate expensive RCF calls by utilising information obtained from past calls) and specific strategies that fine-tune Z3 to the problems that MetiTarski gives it.

Naturally, users would like to handle problems in many more variables. This is the biggest hurdle in any application of RCF decision procedures.

Platzer's treatment of this problem is quite different. He uses KeyMaera, his hybrid system verifier [33]. KeyMaera models a hybrid system that controls the aircraft, while we examine only a part of the continuous dynamics of this system. Even the continuous dynamics are treated differently: KeyMaera has a principle called *differential induction* [31] that can establish properties of the solutions to differential equations without solving them. To prepare these MetiTarski problems, the differential equations must first be solved (perhaps using

[1] On a Mac Pro Dual Quad-core Intel Xeon, 2.8 GHz, with 10GB RAM.

```
fof(airplane_easy,conjecture,
  (! [T,X10,X20,Y10,Y20,D10,D20,E10,E20] :
    (
      (
        0 < T & T < 10 & X10 < -9 & X20 < -1 & Y10 > 10 & Y20 > 10 &
        0.1 < D10 & D10 < 0.15 & 0.1 < D20 & D20 < 0.15 &
        0.1 < E10 & E10 < 0.15 & 0.1 < E20 & E20 < 0.15
      )
      =>
      (
        (X10 - Y10 - 100*D20 - 100*E20 + (100*D20 + 100*E20)*cos(0.01*T)
        + (100*D10 - 100*E10)*sin(0.01*T))^2 +
        (X20 - Y20 + 100*D10 + 100*E10 + (-100*D10 - 100*E10)*cos(0.01*T)
        + (100*D20 - 100*E20)*sin(0.01*T))^2
      )
      > 2
    )
  )
).
include('Axioms/general.ax').
include('Axioms/sin.ax').
include('Axioms/cos.ax').
```

Fig. 1. Aircraft Collision Avoidance

Mathematica), yielding explicit formulas for the aircrafts' trajectories. KeyMaera can verify much larger hybrid systems than MetiTarski can. However, recall that MetiTarski is not specifically designed for verifying this type of example: it is simply a general-purpose theorem prover for the reals.

7 Integration with Proof Assistants

As mentioned before, MetiTarski combines classical resolution with arithmetic simplification, RCF decision procedures and axioms describing the behaviour of various real-valued functions. It returns machine-readable proofs that combine standard resolution steps with these extensions.

If MetiTarski is to be added to an interactive theorem prover as a trusted oracle, little effort is required other than to ensure that all problems submitted to it are first-order with all variables ranging over the real numbers. To reduce the level of trust required, MetiTarski proofs could be broken down into their various elements and reconstructed in Isabelle, leaving only the most intractable steps to oracles.

- Arithmetic simplification in MetiTarski involves little more than reducing polynomials to canonical form and extending the scope of the division operator; these steps should be easy to verify by automation in an LCF-style interactive theorem prover.

- The axioms used by MetiTarski are simply mathematical facts that could, in principle, be developed in any capable interactive theorem prover. Formal developments of the familiar power series expansions already available [16]. However, the theory of continued fractions seems to rest on substantial bodies of mathematics that are yet to be mechanised. Possibly some of these bounds could be verified individually, independently of the general theory.
- The most difficult obstacle is that of the RCF decision procedure. Unfortunately, the algorithms are complicated and we have few implementations to choose from. Until now there has been little interest in procedures that justify their answers. However, Harrison has investigated sum of squares techniques that could produce certificates for such proofs eventually [22].

One fact in our favour is that MetiTarski's entire proof search does not have to be justified, only the final proof, which represents a tiny fraction of the reasoning.

Before undertaking such an integration, it is natural to ask how many potential applications there are. It is sobering to consider that after SMT solvers were integrated with Isabelle, they were left essentially unused [9], and SMT solvers are much more generally applicable than MetiTarski. This integration might be included in a larger project to verify a specific and substantial corpus of continuous mathematics.

8 Conclusion

Research on MetiTarski is proceeding in many directions. Improvements to the use of decision procedures are greatly increasing MetiTarski's scope and power, especially by increasing the number of variables allowed in a problem. Meanwhile, many new kinds of applications are being examined. Integrating MetiTarski with an interactive theorem prover such as Isabelle is not straightforward, but is feasible given motivation and resources.

Acknowledgements. Eva Navarro has made a great effort to give us problems in control engineering to examine. Other members of the team include Paul Jackson and Andrew Sogokon (in Edinburgh), James Bridge, Grant Passmore, William Denman and Zongyan Huang. Denman contributed the collision avoidance example. Bridge and Passmore commented on drafts of this paper, as did Lennart Beringer.

The research was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/C013409/1, EP/I011005/1, EP/I010335/1].

References

1. Abramowitz, M., Stegun, I.A. (eds.): Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. Wiley (1972)
2. Akbarpour, B., Paulson, L.C.: Extending a Resolution Prover for Inequalities on Elementary Functions. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 47–61. Springer, Heidelberg (2007)

3. Akbarpour, B., Paulson, L.C.: MetiTarski: An Automatic Prover for the Elementary Functions. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 217–231. Springer, Heidelberg (2008)
4. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning 44(3), 175–205 (2010)
5. Akbarpour, B., Paulson, L.C.: Towards automatic proofs of inequalities involving elementary functions. In: Cook, B., Sebastiani, R. (eds.) PDPAR: Pragmatics of Decision Procedures in Automated Reasoning, pp. 27–37 (2006)
6. Akbarpour, B., Paulson, L.C.: Applications of MetiTarski in the Verification of Control and Hybrid Systems. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 1–15. Springer, Heidelberg (2009)
7. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A formally verified proof of the prime number theorem. ACM Transactions on Computational Logic 9(1) (2007)
8. Beeson, M.: Automatic generation of a proof of the irrationality of e. JSC 32(4), 333–349 (2001)
9. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT Solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 116–130. Springer, Heidelberg (2011)
10. Bledsoe, W.W.: Non-resolution theorem proving. Artificial Intelligence 9, 1–35 (1977)
11. Bridge, J., Paulson, L.C.: Case splitting in an automatic theorem prover for real-valued special functions. Journal of Automated Reasoning (in press, 2012), http://dx.doi.org/10.1007/s10817-012-9245-6
12. Brown, C.W.: QEPCAD B: a program for computing with semi-algebraic sets using CADs. SIGSAM Bulletin 37(4), 97–108 (2003)
13. Bullen, P.S.: A Dictionary of Inequalities. Longman (1998)
14. Clarke, E., Zhao, X.: Analytica: A theorem prover for Mathematica. Mathematica Journal 3(1), 56–71 (1993)
15. Cuyt, A., Petersen, V., Verdonk, B., Waadeland, H., Jones, W.B.: Handbook of Continued Fractions for Special Functions. Springer (2008)
16. Daumas, M., Muñoz, C., Lester, D.: Verified real number calculations: A library for integer arithmetic. IEEE Trans. Computers 58(2), 226–237 (2009)
17. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symbolic Comp. 5, 29–35 (1988)
18. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
19. Denman, W., Akbarpour, B., Tahar, S., Zaki, M., Paulson, L.C.: Formal verification of analog designs using MetiTarski. In: Biere, A., Pixley, C. (eds.) Formal Methods in Computer Aided Design, pp. 93–100. IEEE (2009)
20. van den Dries, L.: Alfred Tarski's elimination theory for real closed fields. The Journal of Symbolic Logic 53(1), 7–19 (1988)
21. Hardy, R.: Formal Methods for Control Engineering: A Validated Decision Procedure for Nichols Plot Analysis. PhD thesis, University of St Andrews (2006)
22. Harrison, J.: Verifying Nonlinear Real Formulas Via Sums of Squares. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 102–118. Springer, Heidelberg (2007)

23. 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, NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68 (September 2003)
24. Hurd, J.: Metis first order prover (2007), Website at, http://gilith.com/software/metis/
25. Jovanoviċ, D., de Moura, L.: Solving Non-linear Arithmetic. Technical Report MSR-TR-2012-20, Microsoft Research (2012) (accepted to IJCAR 2012)
26. Lassez, J.-L., Maher, M.J.: On Fourier's algorithm for linear arithmetic constraints. Journal of Automated Reasoning 9(3), 373–379 (1992)
27. McLaughlin, S., Harrison, J.: A Proof-Producing Decision Procedure for Real Arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 295–314. Springer, Heidelberg (2005)
28. Mitrinović, D.S., Vasić, P.M.: Analytic Inequalities. Springer (1970)
29. Muñoz, C., Lester, D.R.: Real Number Calculations and Theorem Proving. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 195–210. Springer, Heidelberg (2005)
30. Passmore, G.O., Paulson, L.C., de Moura, L.: Real algebraic strategies for Meti-Tarski proofs. In: Jeuring, J. (ed.) Conferences on Intelligent Computer Mathematics, CICM 2012. Springer (in press, 2012)
31. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Logic Computation 20(1), 309–352 (2010)
32. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
33. Platzer, A., Quesel, J.-D.: KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
34. Ratschan, S.: Efficient solving of quantified inequality constraints over the real numbers. ACM Trans. Comput. Logic 7(4), 723–748 (2006)
35. Ratschan, S., She, Z.: Benchmarks for safety verification of hybrid systems (2008), http://hsolver.sourceforge.net/benchmarks/

Computer-Aided Cryptographic Proofs

Gilles Barthe[1], Juan Manuel Crespo[1], Benjamin Grégoire[2], César Kunz[1,3], and Santiago Zanella Béguelin[4]

[1] IMDEA Software Institute
{Gilles.Barthe,Cesar.Kunz,JuanManuel.Crespo}@imdea.org
[2] Universidad Politécnica de Madrid
[3] INRIA Sophia Antipolis-Méditerranée
Benjamin.Gregoire@inria.fr
[4] Microsoft Research
santiago@microsoft.com

Abstract. EasyCrypt is an automated tool that supports the machine-checked construction and verification of security proofs of cryptographic systems, and that has been used to verify emblematic examples of public-key encryption schemes, digital signature schemes, hash function designs, and block cipher modes of operation. The purpose of this paper is to motivate the role of computer-aided proofs in the broader context of provable security and to illustrate the workings of EasyCrypt through simple introductory examples.

1 Introduction

The rigorous study of cryptographic systems as mathematical objects originates with the landmark article "Communication Theory of Secrecy Systems" [31], in which Shannon defines the notion of perfect secrecy for (symmetric) encryption systems, and shows that it can only be achieved if the size of keys equals or exceeds the size of messages. Shannon's article is often viewed as marking the beginning of modern cryptography, because it was the first to recognize the importance of rigorous mathematical definitions and proofs in the analysis of cryptographic systems.

In contrast to perfect secrecy, which yields unconditional, information-theoretic security, modern cryptography yields conditional guarantees that only hold under computational assumptions. Modern cryptography takes inspiration from complexity theory: rather than considering arbitrary adversaries against the security of cryptographic systems, security is established against adversaries with bounded computational resources. Moreover, the security guarantee itself is probabilistic and is expressed as an upper bound of the probability of an adversary with bounded resources breaking the security of the system. Typically, the computational security of a cryptographic system is proved *by reduction* to one or more assumptions about the hardness of computational problems. This reductionist approach originates from the seminal article "Probabilistic Encryption" [20], in which Goldwasser and Micali elaborate a three-step process for proving the security of a cryptographic system:

```
IND-CPA :
  $(pk, sk) \leftarrow \mathcal{KG}(1^\eta)$;
  $(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $c \leftarrow \mathcal{E}(pk, m_b)$;
  $b' \leftarrow \mathcal{A}_2(c, \sigma)$
```

```
EF-CMA :
  $(pk, sk) \leftarrow \mathcal{KG}(1^\eta)$;
  $(m, s) \leftarrow \mathcal{A}(pk)$

Oracle Sign($m$) :
  $S \leftarrow s :: S$;
  return $\mathcal{S}(sk, m)$
```

```
WCR :
  $k \leftarrow \mathcal{KG}(1^\eta)$;
  $(m_1, m_2) \leftarrow \mathcal{A}()$

Oracle H($m$) :
  return $\mathcal{H}(k, m)$
```

$$\mathbf{Adv}^{(\mathcal{KG},\mathcal{E},\mathcal{D})}_{\mathsf{IND\text{-}CPA}}(\mathcal{A}) \stackrel{\text{def}}{=} |\Pr[\mathsf{IND\text{-}CPA} : b = b'] - 1/2|$$
$$\mathbf{Adv}^{(\mathcal{KG},\mathcal{S},\mathcal{V})}_{\mathsf{EF\text{-}CMA}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr[\mathsf{EF\text{-}CMA} : \mathcal{V}(pk, m, s) \wedge m \notin S]$$
$$\mathbf{Adv}^{(\mathcal{KG},\mathcal{H})}_{\mathsf{WCR}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr[\mathsf{WCR} : \mathcal{H}(k, m_1) = \mathcal{H}(k, m_2) \wedge m_1 \neq m_2]$$

Fig. 1. Experiments corresponding to security notions for various cryptographic constructions (from left to right): *indistinguishability under chosen-plaintext attack* for encryption schemes, *existential unforgeability under chosen-message attack* for signature schemes, and *weak collision-resistance* for keyed hash functions. In these experiments $\mathcal{A}$ denotes an adversary that may have access to oracles; $\mathcal{A}$ has access to a signature oracle $\mathcal{S}(sk, \cdot)$ in experiment EF-CMA and to a hash oracle $\mathcal{H}(k, \cdot)$ in experiment WCR.

1. Formalize precisely the security goal and the adversarial model. A common manner of proceeding is to consider an experiment in which an adversary interacts with a challenger. The challenger sets up and runs the experiment, answers to adversary oracle queries, and determines whether the adversary succeeds. Figure 1 describes experiments corresponding to some typical security notions. Formally, an experiment EXP can be seen as a function that given as input a cryptographic system Π and an adversary $\mathcal{A}$, returns a distribution over some set of output variables. The advantage of an adversary $\mathcal{A}$ in a security experiment EXP, noted $\mathbf{Adv}^{\Pi}_{\mathsf{EXP}}(\mathcal{A})$, is defined in terms of this output distribution.
2. Formalize precisely the assumptions upon which the security of the system relies. Such assumptions assert the practical unfeasibility of solving a computational (or decision) problem believed to be hard. As security goals, they can also be formalized by means of experiments between a challenger and an adversary (an assumption could play the role of a security goal in a lower level proof). Figure 2 describes some assumptions used to realize cryptographic functionalities.
3. Define a cryptographic system Π and give a rigorous proof of its security by exhibiting a reduction from the experiment EXP, corresponding to the security goal, to one or more computational assumptions. Suppose for simplicity that the reductionist proof involves a single assumption, modelled by an experiment EXP′. Broadly construed, the reduction must show that for every efficient adversary $\mathcal{A}$ against EXP, there exists an efficient adversary $\mathcal{B}$ against EXP′ whose advantage is comparable to that of $\mathcal{A}$. In most cases, the proof is constructive and exhibits an adversary $\mathcal{B}$ against EXP′ that uses $\mathcal{A}$ as a sub-routine.

```
DDH₀ :
  $x, y \xleftarrow{\$} [1, \mathsf{ord}(G)]$;
  $b \leftarrow \mathcal{A}(g^x, g^y, g^{xy})$
```

```
DDH₁ :
  $x, y, z \xleftarrow{\$} [1, \mathsf{ord}(G)]$;
  $b \leftarrow \mathcal{A}(g^x, g^y, g^z)$
```

```
OW :
  $(pk, sk) \leftarrow \mathcal{KG}(1^\eta)$;
  $x \xleftarrow{\$} \mathsf{dom}(f)$; $y \leftarrow f(pk, x)$;
  $x' \leftarrow \mathcal{A}(pk, y)$
```

$$\mathbf{Adv}_{\mathsf{DDH}}^{(G,\,g)}(\mathcal{A}) \overset{\text{def}}{=} |\Pr[\mathsf{DDH}_0 : b] - \Pr[\mathsf{DDH}_1 : b]|$$
$$\mathbf{Adv}_{\mathsf{OW}}^{(\mathcal{KG},\,f)}(\mathcal{A}) \overset{\text{def}}{=} \Pr[\mathsf{OW} : f(pk, x') = y]$$

Fig. 2. Experiments corresponding to security assumptions used to realize cryptographic goals: *Decision Diffie-Hellman* problem for a finite cyclic multiplicative group G with generator g (left) and *One-Wayness* of a trapdoor function $(\mathcal{KG}, f)$ (right).

Early works on provable security take an asymptotic approach to capture the notions of efficiency and hardness. In this setting, experiments and assumptions are indexed by a security parameter, typically noted η, which determines the size of objects on which computations are performed (e.g. keys, messages, groups). Asymptotic security equates the class of efficient computations to the class of *probabilistic polynomial-time* algorithms, so that security is established against adversaries whose memory footprint and computation time is bounded by a polynomial on the security parameter. Moreover, in an asymptotic security setting, a problem is considered hard when no efficient adversary can achieve a *non-negligible* advantage as a function of the security parameter. (A function is negligible on η when it is upper-bounded by $1/\eta^c$ for any $c > 0$.)

A more practically relevant approach to cryptographic proofs evaluates quantitatively the efficiency of reductions. This approach, known as practice-oriented provable security or concrete security, originates from the work of Bellare and Rogaway on authentication protocols [10] and the DES block cipher [8]. A typical concrete security proof reducing the security of a construction Π w.r.t. EXP to an assumption EXP' about some object Π', begins by assuming the existence of an adversary $\mathcal{A}$ against EXP that runs within time $t_{\mathcal{A}}$ (and makes at most $q_{\mathcal{A}}$ oracle queries). The proof exhibits a witness for the reduction in the form of an adversary $\mathcal{B}$ against EXP' that uses $\mathcal{A}$ as a sub-routine, and provides concrete bounds for its resources and its advantage in terms of those of $\mathcal{A}$, e.g.:

$$t_B \leq t_A + p(q_{\mathcal{A}})$$
$$\mathbf{Adv}_{\mathsf{EXP}'}^{\Pi'}(\mathcal{B}) \geq \mathbf{Adv}_{\mathsf{EXP}}^{\Pi}(\mathcal{A}) - \epsilon(q_{\mathcal{A}})$$

A concrete security proof can be used to infer sensible values for the parameters (e.g. key size) of cryptographic constructions. Based on an estimate of the resources and advantage of the best known method to solve EXP' (and a conservative bound on $q_{\mathcal{A}}$), one can choose the parameters of Π such that the reduction $\mathcal{B}$ would yield a better method, thus achieving a practical contradiction.

The game-based approach, as popularized by Shoup [32], and Bellare and Rogaway [11], is a methodology to structure reductionist proofs in a way that makes them easier to understand and check. A game-based proof is organized as a tree of games (equivalently, experiments). The root of the tree is the experiment that characterizes the security goal, whereas the leaves are either experiments corresponding to security assumptions or experiments where the probability of an event of interest can be directly bounded. Edges connecting a game G at one level in the tree to its successors $G_1, \ldots, G_n$ correspond to *transitions*; a transition relates the probability of an event in one game to the probability of some, possibly different, event in another game. Put together, these transitions may allow to prove, for example, an inequality of the form

$$\Pr[G : E] \leq a_1 \Pr[G_1 : E_1] + \cdots + a_n \Pr[G_n : E_n]$$

By composing statements derived from the transitions in the tree, one ultimately obtains a bound on the advantage of an adversary against the experiment at the root in terms of the advantage of one of more *concrete* adversaries against assumptions at the leaves.

Whereas games can be formalized in the usual language of mathematics, Bellare and Rogaway [11] model games as probabilistic programs, much like we modelled experiments in Figures 1 and 2. This code-based approach allows giving games a rigorous semantics, and paves the way for applying methods from programming language theory and formal verification to cryptographic proofs. This view was further developed by Halevi [21], who argues that computer-aided verification of cryptographic proofs would be of significant benefit to improve confidence in their correctness, and outlines the design of a computer-aided framework for code-based security proofs.

Verified security [4,5] is an emerging approach to practice-oriented provable security: its primary goal is to increase confidence in reductionist security proofs through their computer-aided formalization and verification, by leveraging state-of-the-art verification tools and programming language techniques. CertiCrypt [5] realizes verified security by providing a fully machine-checked framework built on top of the Coq proof assistant, based on a deep embedding of an extensible probabilistic imperative language to represent games. CertiCrypt implements several verification methods that are proved sound (in Coq) w.r.t. the semantics of programs and inherits the expressive power and the strong guarantees of Coq. Unfortunately, it also inherits a steep learning curve and as a result its usage is time-consuming and requires a high level of expertise. EasyCrypt [4], makes verified security more accessible to the working cryptographer by means of a concise input language and a greater degree of automation, achieved by using off-the-shelf SMT solvers and automated theorem provers rather than an interactive proof assistant like Coq.

Issues with verified security. Verified security is no panacea and inherits several of the issues of provable security and formal proofs in general. We only review briefly some key issues, and refer the interested reader to more detailed reviews of provable security [6,16,30], and formal proofs [22,28]. We stress that these issues do not undermine by any means the importance of verified security.

The first issue regards the interpretation of a verified security proof. As the proof is machine-checked, one can reasonably believe in its correctness without the need to examine the details of the proof. However, a careful analysis of the statement is fundamental to understand the guarantees it provides. In the case of verified security, statements depend on unproven hardness assumptions, which are meaningful only when instantiated with a sensible choice of parameters. Thus, one must consider the security assumptions and convince oneself that they are adequately modelled and instantiated; proofs relying on flawed or inappropriately instantiated assumptions fail to provide any meaningful guarantee. In addition, cryptographic proofs often assume that some functionalities are ideal. As with security assumptions, one must convince oneself that modelling primitives as ideal functionalities is reasonable, and that instantiating these primitives does not introduce subtle attack vectors. Random oracles are a common instance of ideal functionality; in the Random Oracle Model (ROM) [9], some primitives used in a cryptographic system, such as hash functions, are modelled as perfectly random functions, i.e. as maps chosen uniformly from a function space. Proofs in the ROM are considered as providing strong empirical evidence of security, despite some controversy [6,14,18].

The second issue is the level of abstraction in security proofs. Security proofs reason about models rather than implementations. As a result, cryptographic systems, even though supported by a proof of security, may be subject to practical attacks outside the model. Prominent examples of practical attacks are padding oracle attacks [12,26], which exploit information leakage through error handling, and side-channel attacks [13,24,25], which exploit quantitative information such as execution time or memory consumption. There is a growing body of work that addresses these concerns; in particular, leakage-resilient security [19] gives the adversary access to oracles performing side-channel measurements. However, most of the provable security literature, and certainly all of the verified security literature, forego an analysis of side-channels.

Organization of the paper. Section 2 overviews the foundations of EasyCrypt. Subsequent sections focus on examples: One-Time Pad encryption (Section 3), the nested message authentication code NMAC (Section 4), and ElGamal encryption (Section 5). Section 6 reviews some topics deserving more attention.

2 Foundations

This section reviews the foundations of the code-based game-based approach, as implemented by EasyCrypt; more detailed accounts appear in [4,5].

Programming language. Games are represented as programs in the strongly-typed, probabilistic imperative language pWhile:

$\mathcal{C} ::=$	skip	nop
$\mid$	$\mathcal{V} \leftarrow \mathcal{E}$	deterministic assignment
$\mid$	$\mathcal{V} \overset{\$}{\leftarrow} \mathcal{DE}$	probabilistic assignment
$\mid$	$\mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C}$	conditional
$\mid$	$\mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C}$	loop
$\mid$	$\mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \dots, \mathcal{E})$	procedure call
$\mid$	$\mathcal{C};\ \mathcal{C}$	sequence

The language includes deterministic and probabilistic assignments, conditionals, loops, and procedure calls. In the above grammar, $\mathcal{V}$ is a set of variable identifiers, $\mathcal{P}$ a set of procedure names, $\mathcal{E}$ is a set of expressions, and $\mathcal{DE}$ is a set of probabilistic expressions. The latter are expressions that evaluate to distributions from which values can be sampled. An assignment $x \xleftarrow{\$} d$ evaluates the expression d to a distribution μ over values, samples a value according to μ and assigns it to variable x. The base language of expressions (deterministic and probabilistic) can be extended by the user to better suit the needs of the verification goal. The rich base language includes expressions over Booleans, integers, fixed-length bitstrings, lists, finite maps, and option, product and sum types. User-defined operators can be axiomatized or defined in terms of other operators. In the following, we let $\{0,1\}^\ell$ denote the uniform distribution over bitstrings of length ℓ, $\{0,1\}$ the uniform distribution over Booleans, and $[a,b]$ the uniform distribution over the integer interval $[a,b]$.

A program in EasyCrypt is modelled as a set of global variables and a collection of procedures. The language distinguishes between defined procedures, used to describe experiments and oracles, and abstract procedures, used to model adversaries. Quantification over adversaries in cryptographic proofs is achieved by representing them as abstract procedures parametrized by a set of oracles.

Denotational semantics. A pWHILE program c is interpreted as a function $[\![c]\!]$ that maps an initial memory to a sub-distribution over final memories. As pWHILE is a strongly-typed language, a memory is a mapping from variables to values of the appropriate type. When the set of memories is finite, a sub-distribution over memories can be intuitively seen as a mapping assigning to each memory a probability in the unit interval $[0,1]$, so that the sum over all memories is upper bounded by 1. In the general case, we represent a sub-distribution over memories using the *measure monad* of Audebaud and Paulin [1]. Given a program c, a memory m, and an event E, we let $\Pr[c, m : E]$ denote the probability of E in the sub-distribution induced by $[\![c]\!]\ m$; we often omit the initial memory m when it is irrelevant.

Relational program logic. Common reasoning patterns in cryptographic proofs are captured by means of a probabilistic Relational Hoare Logic (pRHL). Its judgments are of the form

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$

where c_1 and c_2 are probabilistic programs, and the pre- and post-conditions Ψ and Φ are relations over memories. Informally, a judgment $\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$ is valid if for every two memories m_1 and m_2 satisfying the pre-condition Ψ, the sub-distributions $[\![c_1]\!]\ m_1$ and $[\![c_2]\!]\ m_2$ satisfy the post-condition Φ. As the post-condition is a relation on memories rather than a relation on sub-distributions over memories, the formal definition of validity relies on a lifting operator, whose definition originates from probabilistic process algebra [17,23].

Relational formulae are represented in EasyCrypt by the grammar:

$$\Psi, \Phi ::= e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \implies \Phi \mid \forall x.\ \Phi \mid \exists x.\ \Phi$$

where e stands for a Boolean expression over logical variables and program variables tagged with either $\langle 1\rangle$ or $\langle 2\rangle$ to denote their interpretation in the left or right-hand side program; the only restriction is that logical variables must not occur free. The special keyword res denotes the return value of a procedure and can be used in the place of a program variable. We write $e\langle i\rangle$ for the expression e in which all program variables are tagged with $\langle i\rangle$. A relational formula is interpreted as a relation on program memories. For example, the formula $x\langle 1\rangle + 1 \leq y\langle 2\rangle$ is interpreted as the relation

$$\Phi = \{(m_1, m_2) \mid m_1(x) + 1 \leq m_2(y)\}$$

Reasoning about probabilities. Security properties are typically expressed in terms of probability of events, and not as pRHL judgments. Pleasingly, one can derive inequalities about probability quantities from valid judgments. In particular, assume that Φ is of the form $A\langle 1\rangle \implies B\langle 2\rangle$, i.e. relates pairs of memories m'_1 and m'_2 such that when m'_1 satisfies the event A, m'_2 satisfies the event B. Then, for any two programs c_1 and c_2 and pre-condition Ψ such that $\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$ is valid, and for any two memories m_1 and m_2 satisfying the pre-condition Ψ, we have $\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B]$. Other forms of pRHL judgments allow to derive more complex inequalities and capture other useful forms of reasoning in cryptographic proofs, including Shoup's Fundamental Lemma [32].

3 Perfect Secrecy of One-Time Pad

Shannon [31] defines perfect secrecy of an encryption scheme by the condition that learning a ciphertext does not change any *a priori* knowledge about the likelihood of messages. In other words, for any given distribution over messages, the distribution over ciphertexts (determined by the random choices of the key generation and encryption algorithms) must be independent of the distribution over messages. Shannon shows that perfect secrecy can only be achieved if the key space is at least as large as the message space, and that the One-Time Pad encryption scheme (also known as Vernam's cipher) is perfectly secret.

For any positive integer ℓ, One-Time Pad is a deterministic symmetric encryption scheme composed of the following triple of algorithms:

Key Generation. The key generation algorithm $\mathcal{KG}$ outputs a uniformly distributed key k in $\{0,1\}^\ell$;

Encryption. Given a key k and a message $m \in \{0,1\}^\ell$, $\mathcal{E}(k,m)$ outputs the ciphertext $c = k \oplus m$ ($\oplus$ denotes bitwise exclusive-or on bitstrings);

Decryption. Given a key k and a ciphertext $c \in \{0,1\}^\ell$, the decryption algorithm outputs the message $m = k \oplus c$.

We represent the a priori distribution over messages by a user-defined probabilisitc operator $\mathcal{M}$. We prove perfect secrecy of One-Time Pad by showing that encrypting a message m sampled according to $\mathcal{M}$ results in a ciphertext distributed uniformly and independently from m. We prove this by showing that the joint distribution of c, m in experiments OTP and Uniform below is the same:

$$\textbf{Game } \mathsf{OTP} : m \xleftarrow{\$} \mathcal{M};\ k \leftarrow \mathcal{KG}();\ c \leftarrow \mathcal{E}(k, m);$$
$$\textbf{Game } \mathsf{Uniform} : m \xleftarrow{\$} \mathcal{M};\ c \xleftarrow{\$} \{0,1\}^{\ell};$$

In a code-based setting, this is captured by the following relational judgment:

$$\models \mathsf{OTP} \sim \mathsf{Uniform} : \mathsf{true} \Rightarrow (c, m)\langle 1\rangle = (c, m)\langle 2\rangle \tag{1}$$

The OTP and Uniform experiments are formalized in EasyCrypt as follows:

```
game OTP = {
  var m : message
  var c : ciphertext
  fun KG() : key = { var k:key = {0,1}^ℓ; return k; }
  fun Enc(k:key, m:message) : ciphertext = { return (k ⊕ m); }
  fun Main() : unit = { var k:key; m = M(); k = KG(); c = Enc(k, m); }
}.

game Uniform = {
  var m : message
  var c : ciphertext
  fun Main() : unit = { m = M(); c = {0,1}^ℓ; }
}.
```

where the types key, message and ciphertext are all synonyms for the type of bitstrings of length ℓ.

The relational judgment (1) is stated and proved in EasyCrypt as follows:

```
equiv Secrecy : OTP.Main ~ Uniform.Main : true ⟹ (c,m)⟨1⟩ = (c,m)⟨2⟩.
proof.
 inline KG, Enc; wp.
 rnd (c ⊕ m); trivial.
save.
```

The proof starts by inlining the definition of the procedures KG and Enc, and applying the wp tactic to compute the relational weakest pre-condition over the deterministic suffix of the resulting programs. This yields the following intermediate goal:

```
pre = true
stmt1 = m = M(); k = {0,1}^ℓ;
stmt2 = m = M(); c = {0,1}^ℓ;
post = (k ⊕ m, m)⟨1⟩ = (c, m)⟨2⟩
```

At this point, we can apply the following pRHL rule for proving equivalence of two uniformly random assignments over the same domain:

$$\frac{f \text{ is a bijection} \qquad \Psi \implies \forall x \in \{0,1\}^{\ell}.\ \Phi\{x/k\langle 1\rangle\}\{f(x)/c\langle 2\rangle\}}{\models k \xleftarrow{\$} \{0,1\}^{\ell} \sim c \xleftarrow{\$} \{0,1\}^{\ell} : \Psi \Rightarrow \Phi}$$

This rule is automated in EasyCrypt by the tactic `rnd`. When given as argument a single expression f as in the above proof script, `rnd` yields a new goal where the post-condition is a conjunction of two formulas universally quantified:

pre = `true`
stmt1 = `m = M();`
stmt2 = `m = M();`
post = $\forall x \in \{0,1\}^{\ell}.\ (x \oplus \mathtt{m}\langle 2\rangle) \oplus \mathtt{m}\langle 2\rangle = x \wedge (x \oplus \mathtt{m}, \mathtt{m})\langle 1\rangle = (x \oplus \mathtt{m}, \mathtt{m})\langle 2\rangle$

The first formula in the post-condition asserts that f, seen as a function of `c`, is an involution (and thus bijective). The second formula is the outcome of substituting x for $\mathtt{k}\langle 1\rangle$ and $f(x) = x \oplus \mathtt{m}\langle 2\rangle$ for $\mathtt{c}\langle 2\rangle$ in the original post-condition. Combining these two formulas under a single quantification results in a more succint goal. A similar rule could be applied to prove an equivalence between the remaining (identical) random assignments. This would leave us with a goal where the statements in both programs are empty and for which it suffices to show that the pre-condition implies the post-condition; the tactic `trivial` does all this automatically using an external solver (e.g. Alt-Ergo [15]) to discharge the resulting proof obligation:

$$\mathsf{true} \implies \forall x, y \in \{0,1\}^{\ell}.\ (x \oplus y) \oplus y = x \wedge (x \oplus y, y) = (x \oplus y, y)$$

4 The NMAC Message Authentication Code

Message Authentication Codes (MACs) are cryptographic algorithms used to provide both authenticity and data integrity in communications between two parties sharing a secret key. At an abstract level, a MAC algorithm M takes as input a key $k \in K$ and a message m, and returns a short bitstring $M(k, m)$—a tag. Given a message m and a key k, a verification algorithm can determine the validity of a tag; for stateless and deterministic MACs, this can be simply done by re-computing the tag. A MAC algorithm is deemed secure if, even after obtaining many valid tags for chosen messages, it is unfeasible to forge a tag for a fresh message without knowing the secret key k. Formally, this can be expressed in terms of the experiment EF-MAC in Figure 3 by requiring that the advantage of an adversary $\mathcal{A}$ that makes at most q queries to a MAC oracle for a freshly sampled key be negligible, where:

$$\mathbf{Adv}^{M}_{\mathsf{EF\text{-}MAC}(q)}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr\left[\mathsf{EF\text{-}MAC} : y = M(k, x) \wedge x \notin X \wedge n \leq q\right]$$

```
Game EF-MAC :              Oracle MAC(x) :
  $k \xleftarrow{\$} K$;     $X \leftarrow x :: X$;
  $X \leftarrow$ nil;        $n \leftarrow n+1$;
  $n \leftarrow 0$;          $z \leftarrow M(k,x)$;
  $(x,y) \leftarrow \mathcal{A}()$   return $z$

Game WCR :                 Oracle F(x) :
  $k \xleftarrow{\$} K$;     $n \leftarrow n+1$;
  $n \leftarrow 0$;          return $F(k,x)$
  $(x_1,x_2) \leftarrow \mathcal{A}()$
```

Fig. 3. Security experiments for MAC Forgery and Weak Collision Resistance

In the remainder of this section we overview the security proof of the NMAC construction [7]. Let ℓ and b be positive integers such that $\ell \leq b$, and let $\mathsf{pad} : \{0,1\}^* \to (\{0,1\}^b)^+$ be an injective function that pads an arbitrary length input message to a positive multiple of b. The NMAC construction transforms a secure fixed input-length MAC $f : \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ into a secure variable input-length MAC:

$$\begin{array}{lcl} \mathsf{NMAC} & : & (\{0,1\}^\ell \times \{0,1\}^\ell) \times \{0,1\}^* \to \{0,1\}^\ell \\ \mathsf{NMAC}((k_1,k_2),m) & \stackrel{\text{def}}{=} & F(k_1, F(k_2,m)) \end{array}$$

where $F(k,m) = f^*(k, \mathsf{pad}(m))$ and $f^* : \{0,1\}^\ell \times (\{0,1\}^b)^* \to \{0,1\}^\ell$ is the function that on input $k \in \{0,1\}^\ell$ and $x = x_1 \cdots x_n$ consisting of n b-bits blocks returns h_n, where $h_0 = k$ and $h_i = f(h_{i-1}, x_i)$ for $1 \leq i \leq n$.

The proof of security for NMAC establishes that it is no more difficult to forge a valid message for NMAC than forging a valid message for the underlying function f, viewed as a MAC, or finding a collision for the keyed function F. Formally, we define Weak Collision-Resistance for F in terms of the experiment WCR shown in Figure 3, and define the advantage of an adversary $\mathcal{A}$ making at most q queries to F as

$$\mathbf{Adv}^{F}_{\mathsf{WCR}(q)}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr\left[\mathsf{WCR} : F(k,x_1) = F(k,x_2) \land x_1 \neq x_2 \land n \leq q\right]$$

Given an arbitrary adversary $\mathcal{A}$ against the security of NMAC, we exhibit two adversaries $\mathcal{A}_F$ and $\mathcal{A}_f$ such that

$$\mathbf{Adv}^{\mathsf{NMAC}}_{\mathsf{EF\text{-}MAC}(q)}(\mathcal{A}) \leq \mathbf{Adv}^{F}_{\mathsf{WCR}(q+1)}(\mathcal{A}_F) + \mathbf{Adv}^{f}_{\mathsf{EF\text{-}MAC}(q)}(\mathcal{A}_f) \qquad (2)$$

Figure 4 shows the tree of games used in the proof. We start from the game encoding an attack against the security of NMAC. We then define another game EF-MAC′ that just introduces a list Y to store the intermediate values of $F(k_2,x)$ computed to answer to oracle queries, and simplifies the definition of NMAC using the identity

$$\mathsf{NMAC}((k_1,k_2),m) \stackrel{\text{def}}{=} f(k_1, \mathsf{pad}(F(k_2,m)))$$

whose validity stems from the fact that the outer application of the function F is on a message of $\ell \leq b$ bits. We prove the following judgment:

$$\models \mathsf{EFMAC} \sim \mathsf{EFMAC}' : \mathsf{true} \Rightarrow \begin{array}{l} (y = \mathsf{NMAC}((k_1,k_2),x) \land x \notin X \land n \leq q)\langle 1\rangle \iff \\ (y = f(k_1, \mathsf{pad}(F(k_2,x))) \land x \notin X \land n \leq q)\langle 2\rangle \end{array}$$

From which we have

$$\mathbf{Adv}^{\mathsf{NMAC}}_{\mathsf{EF\text{-}MAC}_q}(\mathcal{A}) = \Pr\left[\mathsf{EF\text{-}MAC}' : y = f(k_1, \mathsf{pad}(F(k_2, x))) \land x \notin X \land n \leq q\right] \quad (3)$$

We now make a case analysis on whether, when the experiment $\mathsf{EF\text{-}MAC}'$ finishes and $\mathcal{A}$ succeeds, there is a value $x' \in X$ s.t. $F(k_2, x) = F(k_2, x')$ or not. Since we are interested only in executions where $x \notin X$, to make this case analysis it suffices to check whether the value $F(k_2, x)$ is in the list Y.

– If there exists $x' \in X$ such that $F(k_2, x) = F(k_2, x')$, we exhibit an adversary $\mathcal{A}_F$ against the WCR of F that finds a collision. This is trivial: x and x' collide and are necessarily distinct because one belongs to X while the other does not;
– If there is no $x' \in X$ such that $F(k_2, x) = F(k_2, x')$, we exhibit an adversary against the MAC-security of the function f that successfully forges a tag. Indeed, if (x, y) is a forgery for NMAC, then $(\mathsf{pad}(F(k_2, x)), y)$ is a forgery for f.

We prove the following judgments:

$$\begin{aligned}&\models \mathsf{EF\text{-}MAC}' \sim \mathsf{WCR}_F : \mathsf{true} \Rightarrow\\ &\qquad (y = f(k_1, \mathsf{pad}(F(k_2, x))) \land x \notin X \land n \leq q \land F(k_2, x) \in Y)\langle 1\rangle \implies\\ &\qquad (F(k, x_1) = F(k, x_2) \land x_1 \neq x_2 \land n \leq q + 1)\langle 2\rangle\end{aligned}$$

$$\begin{aligned}&\models \mathsf{EF\text{-}MAC}' \sim \mathsf{EF\text{-}MAC}_f : \mathsf{true} \Rightarrow\\ &\qquad (y = f(k_1, \mathsf{pad}(F(k_2, x))) \land x \notin X \land n \leq q \land F(k_2, x) \notin Y)\langle 1\rangle \implies\\ &\qquad (y = f(k, x) \land x \notin X \land n \leq q)\langle 2\rangle\end{aligned}$$

From which follows

$$\begin{aligned}&\Pr\left[\mathsf{EF\text{-}MAC}' : y = f(k_1, \mathsf{pad}(F(k_2, x))) \land x \notin X \land n \leq q\right] \leq\\ &\quad \Pr\left[\mathsf{WCR}_F : F(k, x_1) = F(k, x_2) \land x_1 \neq x_2 \land n \leq q + 1\right] + \qquad (4)\\ &\quad \Pr\left[\mathsf{EF\text{-}MAC}_f : y = f(k, x) \land x \notin X \land n \leq q\right]\end{aligned}$$

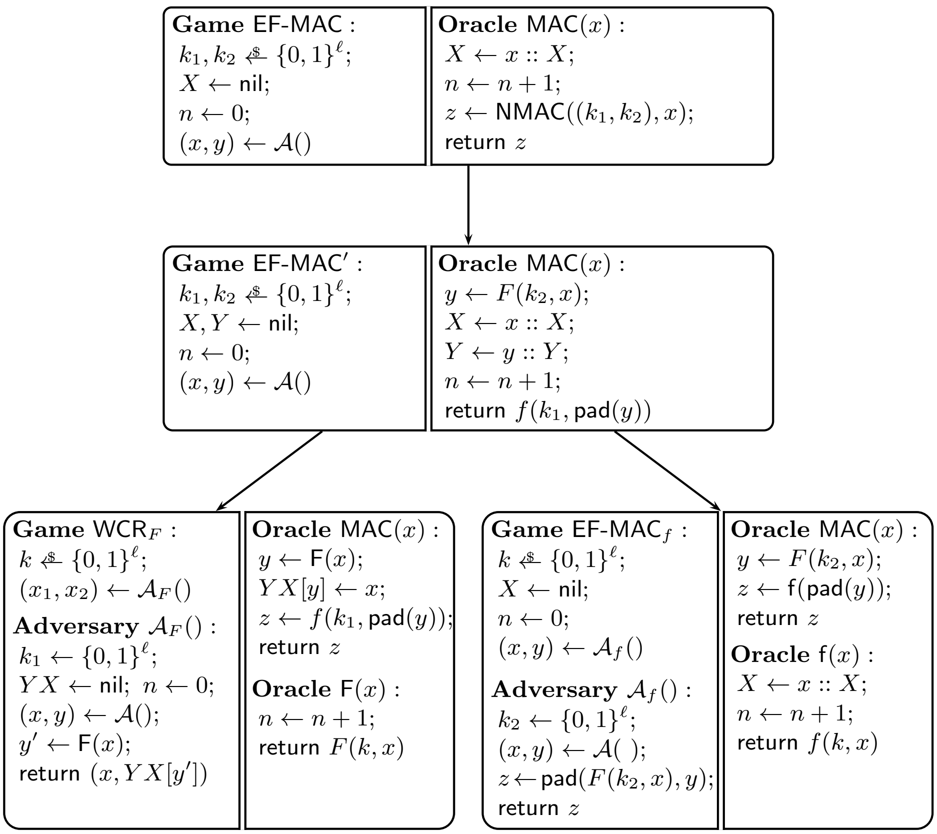
We conclude from (3) and (4) that the bound (2) holds.

We observe that the bound in [7, Theorem 4.1] is off-by-one: the adversary against the WCR-security of F must call the iterated hash function one more time in order to find another value x' that collides with x among the queries made by the adversary against NMAC. Thus, one must assume that the function F is secure against adversaries that make $q + 1$ rather than just q queries.

5 ElGamal Encryption

ElGamal is a public-key encryption scheme based on the Diffie-Hellman key exchange. Given a cyclic group G of order q and a generator g, its key generation, encryption, and decryption algorithms are defined as follows:

$$\begin{aligned}\mathcal{KG}() &\overset{\text{def}}{=} x \xleftarrow{\$} [1, q]; \mathsf{return}\ (g^x, x)\\ \mathcal{E}(\alpha, m) &\overset{\text{def}}{=} y \xleftarrow{\$} [1, q]; \mathsf{return}\ (g^y, \alpha^y \times m)\\ \mathcal{D}(x, (\beta, \zeta)) &\overset{\text{def}}{=} \mathsf{return}\ (\zeta \times \beta^{-x})\end{aligned}$$



```
Game EF-MAC :
  $k_1, k_2 \xleftarrow{\$} \{0,1\}^\ell$;
  $X \leftarrow$ nil;
  $n \leftarrow 0$;
  $(x, y) \leftarrow \mathcal{A}()$

Oracle MAC$(x)$ :
  $X \leftarrow x :: X$;
  $n \leftarrow n + 1$;
  $z \leftarrow$ NMAC$((k_1, k_2), x)$;
  return $z$
```

```
Game EF-MAC$'$ :
  $k_1, k_2 \xleftarrow{\$} \{0,1\}^\ell$;
  $X, Y \leftarrow$ nil;
  $n \leftarrow 0$;
  $(x, y) \leftarrow \mathcal{A}()$

Oracle MAC$(x)$ :
  $y \leftarrow F(k_2, x)$;
  $X \leftarrow x :: X$;
  $Y \leftarrow y :: Y$;
  $n \leftarrow n + 1$;
  return $f(k_1, \mathsf{pad}(y))$
```

```
Game WCR$_F$ :
  $k \xleftarrow{\$} \{0,1\}^\ell$;
  $(x_1, x_2) \leftarrow \mathcal{A}_F()$

Adversary $\mathcal{A}_F()$ :
  $k_1 \leftarrow \{0,1\}^\ell$;
  $YX \leftarrow$ nil; $n \leftarrow 0$;
  $(x, y) \leftarrow \mathcal{A}()$;
  $y' \leftarrow \mathsf{F}(x)$;
  return $(x, YX[y'])$

Oracle MAC$(x)$ :
  $y \leftarrow \mathsf{F}(x)$;
  $YX[y] \leftarrow x$;
  $z \leftarrow f(k_1, \mathsf{pad}(y))$;
  return $z$

Oracle F$(x)$ :
  $n \leftarrow n + 1$;
  return $F(k, x)$
```

```
Game EF-MAC$_f$ :
  $k \xleftarrow{\$} \{0,1\}^\ell$;
  $X \leftarrow$ nil;
  $n \leftarrow 0$;
  $(x, y) \leftarrow \mathcal{A}_f()$

Adversary $\mathcal{A}_f()$ :
  $k_2 \leftarrow \{0,1\}^\ell$;
  $(x, y) \leftarrow \mathcal{A}(\ )$;
  $z \leftarrow \mathsf{pad}(F(k_2, x), y)$;
  return $z$

Oracle MAC$(x)$ :
  $y \leftarrow F(k_2, x)$;
  $z \leftarrow \mathsf{f}(\mathsf{pad}(y))$;
  return $z$

Oracle f$(x)$ :
  $X \leftarrow x :: X$;
  $n \leftarrow n + 1$;
  return $f(k, x)$
```

Fig. 4. Tree of games in the proof of the NMAC construction

Shoup [32] uses ElGamal as a running example to review some interesting points in game-based proofs. We outline a code-based proof of the indistinguishability under chosen-plaintext attacks of ElGamal by reduction to the Decision Diffie-Hellman (DDH) assumption on the underlying group G. The experiments encoding both the security goal and the assumption, were introduced before in Figures 1 and 2 and are instantiated for ElGamal in Figure 5.

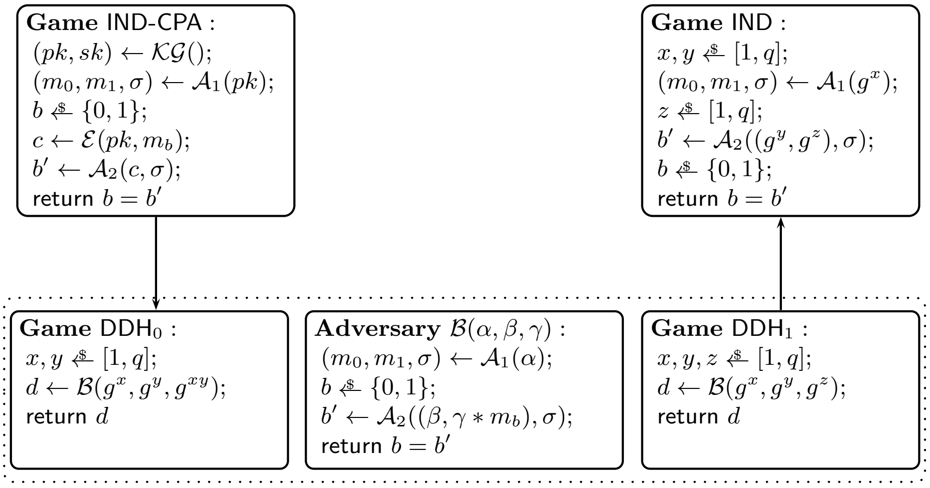
Indistinguishability under chosen-plaintext attacks requires that an efficient adversary cannot distinguish, except with small probability, between two ciphertexts produced from messages of its choice. In the experiment IND-CPA, the challenger samples a fresh pair of keys using the algorithm $\mathcal{KG}$ and gives the public key pk to the adversary, who returns two plaintexts m_0, m_1 of his choice. The challenger then tosses a fair coin b and gives the encryption of m_b under pk back to the adversary, whose goal is to guess which message has been encrypted. We model an IND-CPA adversary $\mathcal{A}$ in EasyCrypt as two unspecified procedures that share state by means of an explicit state variable σ. By keeping the type of this variable abstract, we obtain a generic reduction. Using the keyword res that denotes the return value of a procedure in EasyCrypt, we define the IND-CPA-advantage of $\mathcal{A}$ as in Fig. 1:

$$\mathbf{Adv}^{\mathsf{ElGamal}}_{\mathsf{IND\text{-}CPA}}(\mathcal{A}) \stackrel{\text{def}}{=} \left|\Pr\left[\mathsf{IND\text{-}CPA} : \mathsf{res}\right] - \frac{1}{2}\right|$$

The DDH problem consists in distinguishing between triples of the form (g^x, g^y, g^{xy}) and (g^x, g^y, g^z), where the exponents x, y, z are uniform and independently sampled from the interval $[1..\mathsf{ord}(G)]$. The DDH-advantage of an adversary $\mathcal{B}$ is defined as:

$$\mathbf{Adv}^{(G,\,g)}_{\mathsf{DDH}}(\mathcal{B}) \stackrel{\text{def}}{=} \left|\Pr\left[\mathsf{DDH}_0 : \mathsf{res}\right] - \Pr\left[\mathsf{DDH}_1 : \mathsf{res}\right]\right|$$

Figure 5 presents the overall structure of the reduction, showing a concrete DDH distinguisher $\mathcal{B}$ that achieves exactly the same advantage as an arbitrary IND-CPA adversary $\mathcal{A}$, with constant resource overhead.



```
Game IND-CPA :
  $(pk, sk) \leftarrow \mathcal{KG}()$;
  $(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $c \leftarrow \mathcal{E}(pk, m_b)$;
  $b' \leftarrow \mathcal{A}_2(c, \sigma)$;
  return $b = b'$
```

```
Game IND :
  $x, y \xleftarrow{\$} [1, q]$;
  $(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(g^x)$;
  $z \xleftarrow{\$} [1, q]$;
  $b' \leftarrow \mathcal{A}_2((g^y, g^z), \sigma)$;
  $b \xleftarrow{\$} \{0,1\}$;
  return $b = b'$
```

```
Game DDH_0 :
  $x, y \xleftarrow{\$} [1, q]$;
  $d \leftarrow \mathcal{B}(g^x, g^y, g^{xy})$;
  return $d$
```

```
Adversary $\mathcal{B}(\alpha, \beta, \gamma)$ :
  $(m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(\alpha)$;
  $b \xleftarrow{\$} \{0,1\}$;
  $b' \leftarrow \mathcal{A}_2((\beta, \gamma * m_b), \sigma)$;
  return $b = b'$
```

```
Game DDH_1 :
  $x, y, z \xleftarrow{\$} [1, q]$;
  $d \leftarrow \mathcal{B}(g^x, g^y, g^z)$;
  return $d$
```

Fig. 5. Game-based proof of IND-CPA-security of ElGamal. Games DDH_0 and DDH_1, enclosed in a dotted box, share the same definition for the concrete adversary $\mathcal{B}$.

The proof requires showing the validity of two pRHL judgments. The first judgment relates the experiment IND-CPA to the instantiation of game DDH_0 with the concrete adversary $\mathcal{B}$ defined in Figure 5. We prove that the distribution of the result of the comparison $b = b'$ in game IND-CPA coincides with the distribution of d in game DDH_0, i.e.

$$\models \mathsf{IND\text{-}CPA} \sim \mathsf{DDH}_0 : \mathsf{true} \Rightarrow \mathsf{res}\langle 1\rangle = \mathsf{res}\langle 2\rangle$$

From this, we can derive the equality

$$\Pr\left[\mathsf{IND\text{-}CPA} : \mathsf{res}\right] = \Pr\left[\mathsf{DDH}_0 : \mathsf{res}\right] \tag{5}$$

The second judgment relates the game DDH_1 instantiated with the same adversary $\mathcal{B}$ to a game IND, where the guess b' of the adversary $\mathcal{A}$ no longer depends on the challenge bit b:

$$\models \mathsf{DDH}_1 \sim \mathsf{IND} : \mathsf{true} \Rightarrow \mathsf{res}\langle 1\rangle = \mathsf{res}\langle 2\rangle$$

We state and prove this judgment in EasyCrypt using the following proof script:

```
equiv DDH1_IND : DDH1.Main ∼ IND.Main : true ⟹ res⟨1⟩ = res⟨2⟩.
proof.
 inline B; swap⟨1⟩ 3 2; swap⟨1⟩ [5-6] 2; swap⟨2⟩ 6 -2.
 auto.
 rnd ((z + log(b ? m0 : m1)) % q), ((z - log(b ? m0 : m1)) % q); trivial.
 auto.
 trivial.
save.
```

The `inline` tactic expands calls to procedures by replacing them with their definitions, performing appropriate substitutions and renaming variables if necessary. The tactic `swap` pushes a single instruction or a block of instructions down if its second arguments is positive, or up if it is negative. Dependencies are checked to verify these transformations are semantics-preserving. The tactic `rnd` f, f^{-1} applies the same rule for random assignments that we described in Section 3; except that this time we provide a function f and its inverse f^{-1} by means of justification of its bijectivity. The tactics `auto` and `trivial` implement heuristics to combine simpler tactics. For instance, the above applications of `auto` apply the `wp` transformer and tactics that implement rules for deterministic and random assignments and calls to abstract procedures. This suffices to prove the goal without any user intervention.

It follows from the above judgment that

$$\Pr[\mathsf{DDH}_1 : \mathsf{res}] = \Pr[\mathsf{IND} : \mathsf{res}] \tag{6}$$

The right-hand side of this equality is exactly 1/2, i.e.

$$\Pr[\mathsf{IND} : \mathsf{res}] = \frac{1}{2} \tag{7}$$

This can be proven by direct computation:

```
 claim Fact : IND.Main[res] = 1%r / 2%r by compute.
```

We conclude putting the above equations (5)–(7) together that

$$\mathbf{Adv}^{(G,g)}_{\mathsf{DDH}}(\mathcal{B}) = \mathbf{Adv}^{\mathsf{ElGamal}}_{\mathsf{IND\text{-}CPA}}(\mathcal{A})$$

6 Conclusion

EasyCrypt is a framework for computer-aided cryptographic proofs. It improves confidence in cryptographic systems by delivering formally verified proofs that they achieve their purported goals. In this paper, we have illustrated how EasyCrypt can be used to verify elementary examples. In other works, we have applied EasyCrypt to a range of emblematic examples, including asymmetric encryption schemes [4], signature schemes [33], hash function designs [2,3], and modes of operation for block ciphers. We conclude this article with a review of some topics that deserve further attention. Other topics, not developed below, include the automated synthesis of cryptographic schemes, and the development of more expressive relational program logics for probabilistic programs.

Compositionality. Compositionality and abstraction are fundamental principles in programming language semantics. They are supported by notions such as modules, which are key to structure large software developments. In contrast, it has proved extremely intricate to design general and sound abstraction and compositionality mechanisms for cryptographic proofs. For instance, a recent analysis [29] of the limitations of the indifferentiability framework [27] illustrates the difficulty of instantiating generic proofs to specific constructions. We believe that the code-based approach provides an excellent starting point for developing sound compositional reasoning methods, and that these methods can be incorporated into EasyCrypt.

Automation. EasyCrypt provides automated support to prove the validity of pRHL judgments and to derive inequalities about probability quantities. However, it does not implement any sophisticated mechanism to help users discover or build intermediate games in a game-based proof. It would be interesting to investigate whether one can develop built-in strategies that capture common patterns of reasoning in cryptographic proofs, and generate proof skeletons including the corresponding games and pRHL judgments. A more ambitious goal would be to enhance EasyCrypt with a language for programming strategies, in the way proof assistants such as Coq allow users to program their own tactics.

Certification and mathematical libraries. EasyCrypt was conceived as a front-end to the CertiCrypt framework. In [4], we report on a proof-producing mechanism that converts EasyCrypt files into Coq files that can be machine-checked in the CertiCrypt framework. Certification remains an important objective, although the proof-producing mechanism may fall temporarily out of sync with the development of EasyCrypt. As cryptographic constructions and proofs rely on a wide range of mathematical concepts, the further development of extensive libraries of formalized mathematics is an essential stepping stone towards this goal.

References

1. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in COQ. Sci. Comput. Program. 74(8), 568–589 (2009)
2. Backes, M., Barthe, G., Berg, M., Grégoire, B., Kunz, C., Skoruppa, M., Zanella Béguelin, S.: Verified security of Merkle-Damgård. In: 25rd IEEE Computer Security Foundations Symposium, CSF 2012. IEEE Computer Society (2012)

3. Barthe, G., Grégoire, B., Heraud, S., Olmedo, F., Zanella Béguelin, S.: Verified Indifferentiable Hashing into Elliptic Curves. In: Degano, P., Guttman, J.D. (eds.) POST 2012. LNCS, vol. 7215, pp. 209–228. Springer, Heidelberg (2012)
4. Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: Computer-Aided Security Proofs for the Working Cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
5. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 90–101. ACM, New York (2009)
6. Bellare, M.: Practice-Oriented Provable-Security. In: Okamoto, E. (ed.) ISW 1997. LNCS, vol. 1396, pp. 221–231. Springer, Heidelberg (1998)
7. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
8. Bellare, M., Kilian, J., Rogaway, P.: The Security of Cipher Block Chaining. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 341–358. Springer, Heidelberg (1994)
9. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: 1st ACM Conference on Computer and Communications Security, CCS 1993, pp. 62–73. ACM, New York (1993)
10. Bellare, M., Rogaway, P.: Entity Authentication and Key Distribution. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (1994)
11. Bellare, M., Rogaway, P.: The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006)
12. Bleichenbacher, D.: Chosen Ciphertext Attacks against Protocols Based on the RSA Encryption Standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998)
13. Boneh, D., Franklin, M.: Identity-based encryption from the Weil pairing. SIAM J. Comput. 32(3), 586–615 (2003)
14. Canetti, R., Goldreich, O., Halevi, S.: The random oracle methodology, revisited. J. ACM 51(4), 557–594 (2004)
15. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic combination of congruence closure with solvable theories. Electronic Notes in Theoretical Computer Science 198(2), 51–69 (2008)
16. Damgård, I.: A "proof-reading" of Some Issues in Cryptography. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 2–11. Springer, Heidelberg (2007)
17. Deng, Y., Du, W.: Logical, metric, and algorithmic characterisations of probabilistic bisimulation. Technical Report CMU-CS-11-110, Carnegie Mellon University (March 2011)
18. Dent, A.W.: Fundamental problems in provable security and cryptography. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 364(1849), 3215–3230 (2006)
19. Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, pp. 293–302. IEEE Computer Society, Washington (2008)
20. Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. 28(2), 270–299 (1984)

21. Halevi, S.: A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181 (2005)
22. Harrison, J.: Formal proof – theory and practice. Notices of the American Mathematical Society 55(11), 1395–1406 (2008)
23. Jonsson, B., Yi, W., Larsen, K.G.: Probabilistic extensions of process algebras. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 685–710. Elsevier, Amsterdam (2001)
24. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
25. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
26. Manger, J.: A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 230–238. Springer, Heidelberg (2001)
27. Maurer, U., Renner, R., Holenstein, C.: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer, Heidelberg (2004)
28. Pollack, R.: How to believe a machine-checked proof. In: Twenty-Five Years of Constructive Type Theory: Proceedings of a Congress Held in Venice, October 1995. Oxford Logic Guides, vol. 36. Oxford University Press (1998)
29. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with Composition: Limitations of the Indifferentiability Framework. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer, Heidelberg (2011)
30. Rogaway, P.: Practice-oriented provable security and the social construction of cryptography (2009) (unpublished essay)
31. Shannon, C.: Communication theory of secrecy systems. Bell System Technical Journal 28, 656–715 (1949)
32. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004)
33. Zanella Béguelin, S., Grégoire, B., Barthe, G., Olmedo, F.: Formally certifying the security of digital signature schemes. In: 30th IEEE Symposium on Security and Privacy, S&P 2009, pp. 237–250. IEEE Computer Society, Los Alamitos (2009)

A Differential Operator Approach to Equational Differential Invariants[*]

(Invited Paper)

André Platzer

Computer Science Department, Carnegie Mellon University, Pittsburgh, USA
aplatzer@cs.cmu.edu

Abstract. Hybrid systems, i.e., dynamical systems combining discrete and continuous dynamics, have a complete axiomatization in differential dynamic logic relative to differential equations. Differential invariants are a natural induction principle for proving properties of the remaining differential equations. We study the equational case of differential invariants using a differential operator view. We relate differential invariants to Lie's seminal work and explain important structural properties resulting from this view. Finally, we study the connection of differential invariants with partial differential equations in the context of the inverse characteristic method for computing differential invariants.

1 Introduction

Hybrid systems [1,11] are dynamical systems that combine discrete and continuous dynamics. They are important for modeling embedded systems and cyber-physical systems. Reachability in hybrid systems is neither semidecidable nor co-semidecidable [11]. Nevertheless, hybrid systems have a complete axiomatization relative to elementary properties of differential equations in differential dynamic logic $\mathsf{d}\mathcal{L}$ [18,21]. Using the proof calculus of $\mathsf{d}\mathcal{L}$, the problem of proving properties of hybrid systems reduces to proving properties of continuous systems.

It is provably the case that the only challenge in hybrid systems verification is the need to find invariants and variants [18,21]; the handling of real arithmetic is challenging in practice [27], even if it is decidable in theory [2], but this is not the focus of this paper. According to our completeness results [18,21], we can equivalently focus on either only the discrete or on only the continuous dynamics, because both are equivalently and constructively interreducible, proof-theoretically. Thus, we can equivalently consider the need to prove properties of differential equations as the only challenge in hybrid systems verification. Since the solutions of most differential equations fall outside the usual decidable classes of arithmetic, or do not exist in closed form, the primary means

[*] This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, NSF EXPEDITION CNS-0926181, and under Grant No. CNS-0931985.

for proving properties of differential equations is induction [19]. In retrospect, this is not surprising, because our constructive proof-theoretical alignment [21] shows that every proof technique for discrete systems lifts to continuous systems (and vice versa). Since most verification principles for discrete systems are based on some form of induction, this means that induction is possible for differential equations. *Differential invariants* are such an induction principle. We have introduced differential invariants in 2008 [19], and later refined them to a procedure that computes differential invariants in a fixed-point loop [24,25]. Differential invariants are also related to barrier certificates [29], equational templates [30], and a constraint-based template approach [8]. The structure and theory of general differential invariants has been studied in previous work in detail [23].

In this paper, we focus on the equational case of differential invariants. We show that the equational case of differential invariants and similar approaches is already subsumed by Lie's seminal work [14,15,16,17] in the case of open domains. On open (semialgebraic) domains, Lie's approach gives an equivalence characterization of (smooth) invariant functions. This almost solves the differential invariance generation problem for the equational case completely. It turns out, however, that differential invariants and differential cuts may still prove properties indirectly that the equivalence characterization misses. We carefully illustrate why that is the case. We investigate structural properties of invariant functions and invariant equations. We prove that invariant functions form an algebra and that, in the presence of differential cuts provable invariant equations and valid invariant equations form a chain of differential ideals, whose varieties are generated by a single polynomial, which is the most informative invariant.

Furthermore, we study the connection of differential invariants with partial differential equations. We explain the *inverse characteristic method*, which is the inverse of the usual characteristic method for studying partial differential equations in terms of solutions of corresponding characteristic ordinary differential equations. The inverse characteristic method, instead, uses partial differential equations to study solutions of ordinary differential equations. What may, at first, appear to idiosyncratically reduce the easier problem of ordinary differential equations to the more complicated one of partial differential equations, turns out to be very useful, because it relates the differential invariance problem to mathematically very well-understood partial differential equations.

Even though our results generalize to arbitrary smooth functions, we focus on the polynomial case in this paper, because the resulting arithmetic is decidable.

For background on logic for hybrid systems, we refer to previous work [18,20,22].

2 Differential Dynamic Logic (Excerpt)

Continuous dynamics described by differential equations are a crucial part of hybrid system models. An important subproblem in hybrid system verification is the question whether a system following a (vectorial) differential equation $x' = \theta$ that is restricted to an *evolution domain constraint* region H will always stay in the region F. We represent this by the modal formula $[x' = \theta \,\&\, H]F$. It is true at a state ν if, indeed, a system following $x' = \theta$ from ν will always stay in F at all times (at least as long as the system stays in H). It is false at ν if the system can follow $x' = \theta$ from ν and leave F at some point in time, without having left H at any time. Here, F and H are (quantifier-free) formulas of real arithmetic and $x' = \theta$ is a (vectorial) differential equation, i.e., $x = (x_1, \dots, x_n)$ is a vector of variables and $\theta = (\theta_1, \dots, \theta_n)$ a vector of polynomial terms; for extensions to rational functions, see [19]. In particular, H describes a region that the continuous system cannot leave (e.g., because of physical restrictions of the system or because the controller otherwise switches to another mode of the hybrid system). In contrast, F describes a region which we want to prove that the continuous system $x' = \theta \,\&\, H$ will never leave.

This modal logical principle extends to a full dynamic logic for hybrid systems, called *differential dynamic logic* d$\mathcal{L}$ [18,20,21]. Here we only need first-order logic and modalities for differential equations. For our purposes, it is sufficient to consider the d$\mathcal{L}$ fragment with the following grammar (where x is a vector of variables, θ a vector of terms of the same dimension, and F, H are formulas of (quantifier-free) first-order real arithmetic over the variables x):

$$\phi, \psi \ ::= \ F \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \to \psi \mid \phi \leftrightarrow \psi \mid \forall x\, \phi \mid \exists x\, \phi \mid [x' = \theta \,\&\, H]F$$

A state is a function $\nu : V \to \mathbb{R}$ that assigns real numbers to all variables in the set $V = \{x_1, \dots, x_n\}$. We denote the value of term θ in state ν by $\nu[\![\theta]\!]$. The semantics is that of first-order real arithmetic with the following addition:
$\nu \models [x' = \theta \,\&\, H]F$ iff for each function $\varphi : [0, r] \to (V \to \mathbb{R})$ of some duration r we have $\varphi(r) \models F$ under the following two conditions:

1. the differential equation holds, i.e., for each variable x_i and each $\zeta \in [0, r]$:
$$\frac{\mathsf{d}\, \varphi(t)[\![x_i]\!]}{\mathsf{d}t}(\zeta) = \varphi(\zeta)[\![\theta_i]\!]$$
2. and the evolution domain is respected, i.e., $\varphi(\zeta) \models H$ for each $\zeta \in [0, r]$.

The following simple d$\mathcal{L}$ formula is valid (i.e., true in all states):
$$x > 5 \to [x' = \frac{1}{2}x]x > 0$$
It expresses that x will always be positive if x starts with $x > 5$ and follows $x' = \frac{1}{2}x$ for any period of time.

3 Differential Equations and Differential Operators

In this section, we study differential equations and their associated differential operators. Only properties of very simple differential equations can be proved by working with their solutions, e.g., linear differential equations with constant coefficients that form a nilpotent matrix [18].

Differential Operators. More complicated differential equations need a different approach, because their solutions may not fall into decidable classes of arithmetic, are not computable, or may not even exist in closed form. As a proof technique for advanced differential equations, we have introduced *differential invariants* [19]. Differential invariants turn the following intuition into a formally sound proof procedure. If the vector field of the differential equation always points into a direction where the differential invariant F, which is a logical formula, is becoming "more true" (see Fig. 1), then the system will always stay safe if it initially starts safe. This principle can be understood in a simple but formally sound way in the logic $\mathsf{d}\mathcal{L}$ [19,20]. Differential invariants have been introduced in [19] and later refined to a procedure that computes differential invariants in a fixed-point loop [24]. Instead of our original presentation, which was based on differential algebra, total derivatives, and differential substitution, we take a differential operator approach here. Both views are fruitful and closely related.

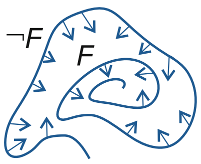


Fig. 1. Differential invariant F

Definition 1 (Lie differential operator). *Let $x' = \theta$ be the differential equation system $x'_1 = \theta_1, \ldots, x'_n = \theta_n$ in vectorial notation. The (Lie)* differential operator *belonging to $x' = \theta$ is the operator $\theta \cdot \nabla$ defined as*

$$\theta \cdot \nabla \stackrel{def}{=} \sum_{i=1}^{n} \theta_i \frac{\partial}{\partial x_i} = \theta_1 \frac{\partial}{\partial x_1} + \cdots + \theta_n \frac{\partial}{\partial x_n} \tag{1}$$

The $\{\frac{\partial}{\partial x_1}, \cdots, \frac{\partial}{\partial x_n}\}$ are partial derivative operators, but can be considered as a basis of the tangent space at x of the manifold on which $x' = \theta$ is defined. The result of applying the differential operator $\theta \cdot \nabla$ to a differentiable function f is

$$(\theta \cdot \nabla) f = \sum_{i=1}^{n} \theta_i \frac{\partial f}{\partial x_i} = \theta_1 \frac{\partial f}{\partial x_1} + \cdots + \theta_n \frac{\partial f}{\partial x_n}$$

The differential operator lifts *conjunctively* to logical formulas F:

$$(\theta \cdot \nabla) F \stackrel{\text{def}}{=} \bigwedge_{(b \sim c) \text{ in } F} \big((\theta \cdot \nabla) b \sim (\theta \cdot \nabla) c\big)$$

This conjunction is over all atomic subformulas $b \sim c$ of F for any operator $\sim \in \{=, \geq, >, \leq, <\}$. In this definition, we assume that formulas use dualities like $\neg(a \geq b) \equiv a < b$ to avoid negations and the operator $\neq$ is handled in a special way; see previous work for a discussion [19,22]. The functions and terms in f and F need to be sufficiently smooth for the partial derivatives to be defined and enjoy useful properties like commutativity of $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$. This is the case for polynomials, which are arbitrarily smooth (C^∞).

Since the differential operator $\theta \cdot \nabla$ is a combination of the total derivative and differential substitution, we have elsewhere [19,22] denoted the result $(\theta \cdot \nabla)F$ of applying $\theta \cdot \nabla$ to a logical formula F by ${F'}^{\theta}_{x'}$. The latter notation is also appropriate, because $(\theta \cdot \nabla)F \equiv {F'}^{\theta}_{x'}$ can, indeed, be formed by taking the total derivative F' and then substituting in the right-hand side θ of the differential equation to replace its left-hand side x', the result of which is denoted ${F'}^{\theta}_{x'}$. It is insightful [19] to give a semantics to F', because that is the key to proving advanced differential transformations [19], but beyond the scope of this paper. We refrain from using this alternative notation in this paper, because we want to emphasize the differential operator nature of the combined derivative and differential substitution. In this notation, our differential induction proof rule [19] is:

$$\text{(DI)}\ \frac{H \rightarrow (\theta \cdot \nabla)F}{F \rightarrow [x' = \theta \,\&\, H]F}$$

This *differential induction* rule is a natural induction principle for differential equations. The difference compared to ordinary induction for discrete loops is that the evolution domain constraint H is assumed in the premise (because the continuous evolution is not allowed to leave its evolution domain constraint) and that the induction step uses the differential formula $(\theta \cdot \nabla)F$ corresponding to formula F and the differential operator $\theta \cdot \nabla$ belonging to the differential equation $x' = \theta$ instead of a statement that the loop body preserves the invariant. Intuitively, the *differential formula* $(\theta \cdot \nabla)F$ captures the infinitesimal change of formula F over time along $x' = \theta$, and expresses the fact that F is only getting more true when following the differential equation $x' = \theta$. The semantics of differential equations is defined in a mathematically precise but computationally intractable way using analytic differentiation and limit processes at infinitely many points in time. The key point about differential invariants is that they replace this precise but computationally intractable semantics with a computationally effective use of a differential operator. The valuation of the resulting computable formula $(\theta \cdot \nabla)F$ along differential equations coincides with analytic differentiation [19]. The term $(\theta \cdot \nabla)p$ characterizes how p changes with time along a solution of $x' = \theta$.

Lemma 2 (Derivation lemma). *Let $x' = \theta \,\&\, H$ be a differential equation with evolution domain constraint H and let $\varphi : [0, r] \rightarrow (V \rightarrow \mathbb{R})$ be a corresponding solution of duration $r > 0$. Then for all terms p and all $\zeta \in [0, r]$:*

$$\frac{\mathsf{d}\,\varphi(t)[\![p]\!]}{\mathsf{d}t}(\zeta) = \varphi(\zeta)[\![(\theta \cdot \nabla)p]\!] \ .$$

Proof. This lemma can either be shown directly or by combining the derivation lemma [19, Lemma 1] with differential substitution [19, Lemma 2]. □

The rule DI for differential invariance is computationally very attractive, because it replaces the need to reason about complicated solutions of differential equations with simple symbolic computation and arithmetic on terms that are formed by differentiation, and, hence, have lower degree. The primary challenge, however, is to find a suitable F for a proof.

Equational Differential Invariants. General formulas with propositional combinations of equations and inequalities can be used as differential invariants. For the purposes of this paper, we focus on the equational case in more detail, which is the following special case of DI:

$$(DI_=)\ \frac{H\to(\theta\cdot\nabla)p=0}{p=0\to[x'=\theta\,\&\,H]p=0}$$

This equational case of differential invariants turns out to be a special case of Lie's seminal work on what are now called Lie groups [15,16]. Since θ and p are (sufficiently) smooth, we can capture Lie's theorem [17, Proposition 2.6] as a d$\mathcal{L}$ proof rule to make the connection to $DI_=$ more apparent.

Theorem 3 (Lie [15,16]). *Let $x'=\theta$ be a differential equation system and H a* domain*, i.e., a first-order formula of real arithmetic characterizing an open set. The following proof rule is a sound global equivalence rule, i.e., the conclusion is valid if and only if the premise is.*

$$(DI_c)\ \frac{H\to(\theta\cdot\nabla)p=0}{\forall c\,\big(p=c\to[x'=\theta\,\&\,H]p=c\big)}$$

That is, the following d$\mathcal{L}$ *axiom is sound, i.e., all of its instances valid*

$$\forall x\,\forall c\,\big(p=c\to[x'=\theta\,\&\,H]p=c\big)\leftrightarrow\forall x\,(H\to(\theta\cdot\nabla)p=0)$$

Proof (Sketch). We only sketch a proof for the soundness direction of DI_c and refer to [15,16,17,19] for a full proof. Suppose there was a ζ with $\varphi(\zeta)[\![p]\!]\neq\varphi(0)[\![p]\!]$, then, by mean-value theorem, there is a $\xi<\zeta$ such that, when using Lemma 2:

$$0\neq\varphi(\zeta)[\![p]\!]-\varphi(0)[\![p]\!]=(\zeta-0)\frac{\mathsf{d}\varphi(t)[\![p]\!]}{\mathsf{d}t}(\xi)=\zeta\varphi(\xi)[\![(\theta\cdot\nabla)p]\!]$$

Thus, $\varphi(\xi)[\![(\theta\cdot\nabla)p]\!]\neq 0$, which contradicts the premise (when $H\equiv \mathit{true}$). □

Note that domains are usually assumed to be connected. We can reason separately about each connected component of H, which are only finitely many, because our domains are first-order definable in real-closed fields [31]. Observe that the conclusion of DI_c implies that of $DI_=$ by instantiating c with 0.

Corollary 4 (Decidability of invariant polynomials). *It is decidable, whether a polynomial p with real algebraic coefficients is an invariant function for a given $x'=\theta$ on a (first-order definable) domain H (i.e., the conclusion of DI_c holds). In particular, the set of polynomials with real algebraic coefficients that are invariant for $x'=\theta$ is recursively enumerable.*

This corollary depends on the fact that real algebraic coefficients are countable. A significantly more efficient version of the recursive enumerability is obtained when using symbolic parameters as coefficients in a polynomial p of increasing degree and using the fact that the equivalence in Theorem 3 is valid for each choice of p. In particular, when p is a polynomial with a vector a of symbolic parameters, then, by Theorem 3, the following d$\mathcal{L}$ formula is valid

$$\exists a\,\forall x\,\forall c\,\big(p=c\to[x'=\theta\,\&\,H]p=c\big)\leftrightarrow\exists a\,\forall x\,(H\to(\theta\cdot\nabla)p=0)\qquad(2)$$

The right-hand side is decidable in the first-order theory of real-closed fields [31]. Hence, so is the left-hand side, but the approach needs to be refined to be useful.

This includes a logical reformulation of the so-called *direct method*, where the user guesses an *Ansatz* p, e.g., as a polynomial with symbolic parameters a instead of concrete numbers as coefficients, and these parameters are instantiated as needed during the attempt to prove invariance of p. In d$\mathcal{L}$, we do not need to instantiate parameters a, because it is sufficient to prove existence, for which there are corresponding d$\mathcal{L}$ proof principles [18]. Other constraints on p need to be considered, however, e.g., that $p = 0$ holds in the initial state and $p = 0$ implies the desired postcondition. Otherwise, the instantiation of a that yields the zero polynomial would be a solution for (2), just not a very insightful one. For example, let d$\mathcal{L}$ formula A characterize the initial state and d$\mathcal{L}$ formula B be the postcondition for a continuous system $x' = \theta \,\&\, H$. Then validity of the following (arithmetic) formula

$$\exists a\, \forall x\, ((H \rightarrow (\theta \cdot \nabla)p = 0) \land (A \rightarrow p = 0) \land (H \land p = 0 \rightarrow B) \tag{3}$$

implies validity of the d$\mathcal{L}$ formula

$$A \rightarrow [x' = \theta \,\&\, H]B$$

Formula (3) is decidable if A and B are first-order real arithmetic formulas. Otherwise, the full d$\mathcal{L}$ calculus is needed to prove (3). Existential quantifiers for parameters can be added in more general ways to d$\mathcal{L}$ formulas with full hybrid systems dynamics to obtain an approach for generating invariants for proving more general properties of hybrid systems [24,25]. The *Ansatz* p can also be varied automatically by enumerating one polynomial with symbolic coefficients for each (multivariate) degree. This direct method can be very effective, and is related to similar approaches for deciding universal real-closed field arithmetic [27], but, because of the computational cost of real arithmetic [7,4], stops to be efficient for complicated high-dimensional problems. In this paper, we analyze the invariance problem further to develop a deeper understanding of its challenges and ways of solving it.

Since DI$_c$ is an equivalence, Theorem 3 and its corollary may appear to solve the invariance problem (for equations) completely. Theorem 3 is a very powerful result, but there are still many remaining challenges in solving the invariance problem as we illustrate in the following.

Counterexample 5 (Deconstructed aircraft). The following d$\mathcal{L}$ formula is valid. It is a much simplified version of a formula proving collision freedom for an air traffic control maneuver [19,26]. We have transformed the differential equations to a physically less interesting case that is notationally simpler and still exhibits similar technical phenomena as those that occur in air traffic control verification.

$$x^2 + y^2 = 1 \land e = x \rightarrow [x' = -y, y' = e, e' = -y](x^2 + y^2 = 1 \land e = x) \tag{4}$$

This d$\mathcal{L}$ formula expresses that an aircraft with position (x, y) will always be safely separated from the origin $(0, 0)$, here, by exactly distance 1 to make things easier. Formula (4) also expresses that the aircraft always is in a compatible y-direction e compared to its position (x, y). In the full aircraft scenario, there is more than one aircraft, each aircraft has more than one direction variable, the relation of the directions to the positions is more complex, and the distance of the aircraft to each other is not fixed at 1, it can be any distance bigger than a protected zone, etc. Yet the basic mathematical phenomena when analyzing (4) are similar to those for full aircraft [19,26], which is why we focus on (4) for notational simplicity. Unfortunately, when we try to prove the valid $\mathsf{d}\mathcal{L}$ formula (4) by a Lie-type differential invariance argument, the proof fails

$$\begin{array}{rl}
 & \text{not valid} \\ \hline
 & -2xy + 2ey = 0 \\ \hline
 & (-y)2x + e2y = 0 \land -y = -y \\ \hline
 & -y\frac{\partial(x^2+y^2)}{\partial x} + e\frac{\partial(x^2+y^2)}{\partial y} = 0 \land -y\frac{\partial e}{\partial e} = -y\frac{\partial x}{\partial x} \\ \hline
\text{DI} & x^2 + y^2 = 1 \land e = x \rightarrow [x' = -y, y' = e, e' = -y](x^2 + y^2 = 1 \land e = x)
\end{array}$$

This is, at first, surprising, since Theorem 3 is an equivalence, but the conclusion (4) is valid and, yet, the proof does not close. On second thought, the postcondition is a propositional combination of equations instead of the single equation assumed in DI_c and $DI_=$. This discrepancy might have caused Theorem 3 to fail. That is not the issue, however, because we have shown that the deductive power of equational differential invariants equals the deductive power of propositional combinations of equations [19, Proposition 1][23, Proposition 5.1]. That is, every formula that is provable using propositional combinations of equations as differential invariants is provable with single equational differential invariants.

Proposition 6 (Equational deductive power [19,23]). *The deductive power of differential induction with atomic equations is identical to the deductive power of differential induction with propositional combinations of polynomial equations: That is, each formula is provable with propositional combinations of equations as differential invariants iff it is provable with only atomic equations as differential invariants.*

Using the construction of the proof of Proposition 6 on the situation in Counterexample 5, we obtain the following counterexample.

Counterexample 7 (Deconstructed aircraft atomic). The construction in the (constructive) proof of Proposition 6 uses an equivalence, here, the following:

$$x^2 + y^2 = 1 \land e = x \equiv (x^2 + y^2 - 1)^2 + (e - x)^2 = 0$$

The right-hand side of the equivalence is a valid invariant and now a single polynomial as assumed in Theorem 3, but DI_c and $DI_=$ still do not prove it, even though the desired conclusion is valid (because it follows from (4) by axiom K and Gödel's generalization [21]):

$$\begin{array}{ll}
 & \text{not valid} \\ \hline
 & 2(x^2+y^2-1)(-2yx+2ey)=0 \\ \hline
 & 2(x^2+y^2-1)(-y2x+e2y)+2(e-x)(-y-(-y))=0 \\ \hline
 & (-y\frac{\partial}{\partial x}+e\frac{\partial}{\partial y}-y\frac{\partial}{\partial e})\big((x^2+y^2-1)^2+(e-x)^2\big)=0 \\ \hline
\text{DI} & (x^2+y^2-1)^2+(e-x)^2=0 \rightarrow [x'=-y,y'=e,e'=-y](x^2+y^2-1)^2+(e-x)^2=0
\end{array}$$

How can that happen? And what can we do about it? The key to understanding this is the observation that we *could* close the above proof if only we knew that $e = x$, which is part of the invariant we are trying to prove in this proof attempt. Note that the relation of the variables in the air traffic control maneuver is more involved than mere identity. In that case, a similar relation of the state variables still exists, involving the angular velocity, positions, and multidimensional directions of the aircraft. This relation is crucial for a corresponding proof; see previous work [19,26].

We could close the proof attempt in Counterexample 7 if only we could assume in the premise the invariant F that we are trying to prove. A common mistake is to suspect that F (or the boundary of F) could, indeed, be assumed in the premise when proving invariance of F along differential equations. That would generally be unsound even though it has been suggested [28,8].

Counterexample 8 (No recursive assumptions). The following counterexample shows that it is generally unsound to assume invariants like $F \equiv x^2 - 6x + 9 = 0$ in the antecedent of the induction step for equational differential invariants

$$\begin{array}{l}
\text{unsound} \\ \hline
x^2-6x+9=0 \rightarrow y2x-6y=0 \\ \hline
x^2-6x+9=0 \rightarrow y\frac{\partial(x^2-6x+9)}{\partial x}-x\frac{\partial(x^2-6x+9)}{\partial y}=0 \\ \hline
x^2-6x+9=0 \rightarrow [x'=y,y'=-x]x^2-6x+9=0
\end{array}$$

We have previously identified [19] conditions under which F can still be assumed soundly in the differential induction step. Those conditions include the case where F is open or where the differential induction step can be strengthened to an open condition with strict inequalities. Unfortunately, these cases do not apply to equations, which are closed and rarely satisfy strict inequalities in the differential induction step. In particular, we cannot use those to close the proof in Counterexample 7.

Differential Cuts. As an alternative, we have introduced differential cuts [19]. *Differential cuts* [19] are a fundamental proof principle for differential equations. They can be used to strengthen assumptions in a sound way:

$$(\text{DC})\ \frac{F \rightarrow [x'=\theta\,\&\,H]C \qquad F \rightarrow [x'=\theta\,\&\,(H\land C)]F}{F \rightarrow [x'=\theta\,\&\,H]F}$$

The differential cut rule works like a cut, but for differential equations. In the right premise, rule DC restricts the system evolution to the subdomain $H \land C$

of H, which restricts the system dynamics to a subdomain but this change is a pseudo-restriction, because the left premise proves that the extra restriction C on the system evolution is an invariant anyhow (e.g. using rule DI). Note that rule DC is special in that it changes the dynamics of the system (it adds a constraint to the system evolution domain region that the resulting system is never allowed to leave), but it is still sound, because this change does not reduce the reachable set. The benefit of rule DC is that C will (soundly) be available as an extra assumption for all subsequent DI uses on the right premise of DC. In particular, the differential cut rule DC can be used to strengthen the right premise with more and more auxiliary differential invariants C that cut down the state space and will be available as extra assumptions to prove the right premise, once they have been proven to be differential invariants in the left premise.

Using differential cuts repeatedly in a process called *differential saturation* has turned out to be extremely useful in practice and even simplifies the invariant search, because it leads to several simpler invariants to find and prove instead of a single complex property [24,25,20]. Differential cuts helped us find proofs for collision avoidance protocols for aircraft [19,26]. Following the same principle in the simplified case of deconstructed aircraft, we finally prove the separation property (4) by a differential cut. The differential cut elimination hypothesis, i.e., whether differential cuts are necessary, has been studied in previous work [23] and will be discussed briefly later.

Example 9 (Differential cuts help separate aircraft). With the help of a differential cut by $e = x$, we can now prove the valid $\mathsf{d}\mathcal{L}$ formula (4), which is a deconstructed variant of how safe separation of aircraft can be proved. For layout reasons, we first show the left premise resulting from DC

$$\mathrm{DC}\,\dfrac{\mathrm{DI}\,\dfrac{\dfrac{\mathbb{R}\,\dfrac{*}{-y=-y}}{-y\frac{\partial e}{\partial e}=-y\frac{\partial x}{\partial x}}}{e=x\to[x'=-y,y'=e,e'=-y]e=x} \qquad \triangleright}{x^2+y^2=1\land e=x\to[x'=-y,y'=e,e'=-y](x^2+y^2=1\land e=x)}$$

and then show the proof of the right premise of DC resulting from the hidden branch (indicated by $\triangleright$ above):

$$\mathrm{DI}\,\dfrac{\dfrac{\dfrac{\mathbb{R}\,\dfrac{*}{e=x\to-2yx+2xy=0}}{e=x\to(-y)2x+e2y=0}}{e=x\to-y\frac{\partial(x^2+y^2)}{\partial x}+e\frac{\partial(x^2+y^2)}{\partial y}=0}}{x^2+y^2=1\land e=x\to[x'=-y,y'=e,e'=-y\,\&\,e=x](x^2+y^2=1\land e=x)}$$

Finally, we have a proof of (4) even if it took more than Theorem 3 to prove it.

Another challenge in invariance properties of differential equations the following. Theorem 3 is sufficient, i.e., the premise of DI_c implies the conclusion even if H is not a domain. But the converse direction of necessity may stop to hold, because the conclusion might hold only because all evolutions immediately leave the evolution domain H.

Counterexample 10 (Equivalence requires domain). The following counterexample shows that the equivalence of DI_c requires H to be a domain

$$\cfrac{\cfrac{\cfrac{\text{not valid}}{y=0 \to 2=0}}{y=0 \to (2\frac{\partial}{\partial x} + 3\frac{\partial}{\partial y})x = 0}}{\forall c \left(x=c \to [x'=2, y'=3 \,\&\, y=0]x=c\right)}$$

Here, the (closed) restriction $y=0$ has an empty interior and $y'=3$ leaves it immediately. The fact that the evolution leaves $y=0$ immediately is the only reason why $x=c$ is an invariant, which would otherwise not be true, because $x'=2$ leaves $x=c$ when evolving for any positive duration. That is why the above premise is not valid even if the conclusion is. Consequently, DI_c can miss some invariants if H is not a domain. Similar phenomena occur when H has a non-empty interior but is not open.

In the proof of Example 9, after the differential cut (DC) with $e=x$, the refined evolution domain constraint is not a domain anymore, which may appear to cause difficulties in the reasoning according to Counterexample 10. Whether evolution domain restrictions introduced by differential cuts are domains, however, is irrelevant, because the left premise of DC just proved that the differential equation (without the extra constraint C) never leaves C, which turns C into a manifold on which differentiation is well-defined and Lie's theorem applies.

Example 11 (Indirect single proof proof of aircraft separation). We had originally conjectured in 2008 [19] that the differential cuts as used in Example 9 and for other aircraft dynamics are necessary to prove these separation properties. We recently found out, however, that this is not actually the case [23]. The following proof of (4) uses a single differential induction step and no differential cuts:

$$\mathrm{DI}\;\cfrac{\mathbb{R}\;\cfrac{\cfrac{*}{-y2e + e2y = 0 \land -y = -y}}{-y\frac{\partial(e^2+y^2)}{\partial e} + e\frac{\partial(e^2+y^2)}{\partial y} = 0 \land -y\frac{\partial e}{\partial e} = -y\frac{\partial x}{\partial x}}}{e^2+y^2=1 \land e=x \to [x'=-y, y'=e, e'=-y](e^2+y^2=1 \land e=x)}$$

Using the construction in Proposition 6, a corresponding proof uses only a single equational invariant to prove (4):

$$\mathrm{DI}\;\cfrac{\mathbb{R}\;\cfrac{\cfrac{*}{2(e^2+y^2-1)(-y2e+e2y) + 2(e-x)(-y-(-y)) = 0}}{(-y\frac{\partial}{\partial x} + e\frac{\partial}{\partial y} - y\frac{\partial}{\partial e})\left((e^2+y^2-1)^2 + (e-x)^2\right) = 0}}{(e^2+y^2-1)^2 + (e-x)^2 = 0 \to [x'=-y, y'=e, e'=-y](e^2+y^2-1)^2 + (e-x)^2 = 0}$$

Thus, DC and domain restrictions are not critical for proving (4). Observe, however, that the indirect proof of (4) in Example 11 worked with a single equational differential invariant and recall that the same formula was not provable directly in Counterexample 5. Thus, even when the evolution domain (here *true*) is a domain and the phenomena illustrated in Counterexample 10 are not an issue, indirect proofs with auxiliary invariants may succeed even if the direct use of DI_c fails. This makes Theorem 3 incomplete and invariant generation challenging.

Before we illustrate the reasons for this difference in the next section, we briefly show that the same phenomenon happens for the actual aircraft dynamics, not just the deconstructed aircraft-type dynamics.

Example 12 (Aircraft). We abbreviate $d_1^2 + d_2^2 = \omega^2 p^2 \wedge d_1 = -\omega x_2 \wedge d_2 = \omega x_1$ by F, which is equivalent to the condition $x_1^2 + x_2^2 = p^2 \wedge d_1 = -\omega x_2 \wedge d_2 = \omega x_1$ for safe separation by distance p of the aircraft (x_1, x_2) from the origin $(0,0)$, when the aircraft flies in a roundabout in its current direction (d_1, d_2) with angular velocity $\omega \neq 0$. We prove invariance of F for an aircraft:

$$\dfrac{\dfrac{\dfrac{*}{2d_1(-\omega d_2) + 2d_2\omega d_1 = 0 \wedge -\omega d_2 = -\omega d_2 \wedge \omega d_1 = \omega d_1}\,\mathbb{R}}{2d_1 d_1' + 2d_2 d_2' = 0 \wedge d_1' = -\omega x_2' \wedge d_2' = \omega x_1'}}{F \wedge \omega \neq 0 \rightarrow [x_1' = d_1, x_2' = d_2, d_1' = -\omega d_2, d_2' = \omega d_1]F}\,\mathrm{DI}$$

The proof for collision freedom of an aircraft (x_1, x_2) in direction (d_1, d_2) from an aircraft (y_1, y_2) flying in direction (e_1, e_2) is similar to that in [19].

While differential cuts have, thus, turned out not to be required (though still practically useful) for these aircraft properties, differential cuts are still crucially necessary to prove other systems. We have recently shown that differential cuts increase the deductive power fundamentally [23]. That is, unlike in the first-order case, where Gentzen's cut elimination theorem [6] proves that first-order cuts can be eliminated, we have refuted the *differential cut elimination hypothesis*, by proving that some properties of differential equations can only be proved with a differential cut, not without.

Theorem 13 (Differential cut power [23]). *The deductive power with differential cuts (rule DC) exceeds the deductive power without differential cuts.*

We refer to previous work [23] for details on the differential cut elimination hypothesis [19], the proof of its refutation [23], and a complete investigation of the relative deductive power of several classes of differential invariants.

4 Invariant Equations and Invariant Functions

In this section, we study invariant equations and the closely related notion of invariant functions. The conclusion of rule DI_c expresses that the polynomial term p is an invariant function of the differential equation $x' = \theta$ on domain H:

Definition 14 (Invariant function). *The function p is an* invariant function *of the differential equation $x' = \theta$ on H iff*

$$\vDash \forall c\,\big(p = c \to [x' = \theta \,\&\, H]p = c\big)$$

That is, an invariant function p is one whose value $p(x(t))$ is constant along all solutions $x(t)$, as a function of time t, of the differential equation $x' = \theta$ within the domain H, i.e., $p(x(t)) = p(x(0))$ for all t. Rule DI_c provides a way to prove that p is an invariant function. A closely related notion is the following.

Definition 15 (Invariant equation). *For a function p, the equation $p = 0$ is an* invariant equation *of the differential equation $x' = \theta$ on H iff*

$$\vDash p = 0 \to [x' = \theta \,\&\, H]p = 0$$

Synonymously, we say that $p = 0$ is an equational invariant or that the variety $V(p)$ is an *invariant variety* of $x' = \theta \,\&\, H$. For a set S of functions (or polynomials), $V(S)$ is the *variety* of zeros of S:

$$V(S) \overset{\text{def}}{=} \{a \in \mathbb{R}^n : f(a) = 0 \text{ for all } f \in S\}$$

For a single function or polynomial p, we write $V(p)$ for $V(\{p\})$. Varieties of sets of polynomials are a fundamental object of study in algebraic geometry [3,10]. Rule $DI_=$ provides a way to prove that $p = 0$ is an invariant equation.

What is, at first, surprising, is that the premise of rule $DI_=$ does not depend on the constant term of the polynomial p. However, a closer look reveals that the premises of $DI_=$ and DI_c are equivalent, and, hence, rule $DI_=$ actually proves that p is an invariant function, not just that $p = 0$ is an equational invariant. Both notions of invariance are closely related but different. If p is an invariant function, then $p = 0$ is an equational invariant [17], but not conversely, since not every level set of p has to be invariant if $p = 0$ is invariant; compare Fig. 2 to general differential invariant Fig. 1.

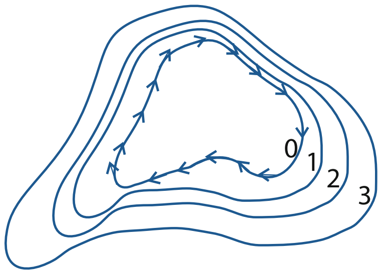


Fig. 2. Invariant equations $p = c$ for levels c of invariant function p

Lemma 16 (Relation of invariant functions and invariant equations). *A (smooth) polynomial p is an invariant function of $x' = \theta \,\&\, H$ iff, for every $c \in \mathbb{R}$, $p = c$ is an invariant equation of $x' = \theta \,\&\, H$. In this case, if c is a constant that denotes the value of p at the initial state, then $p = c$ and $p = 0$ are invariant equations. Conversely, if $p = 0$ is an equational invariant then the product $I_{p=0}p$ is an invariant function (not necessarily C^1, i.e., continuously differentiable). If c is a fresh variable and $p = c$ an invariant equation of $x' = \theta, c' = 0 \,\&\, H$, then p is an invariant function of $x' = \theta \,\&\, H$ and $x' = \theta, c' = 0 \,\&\, H$.*

Proof. By definition. Recall that the characteristic or indicator function of $p = 0$ is defined as $I_{p=0}(x) = 1$ if $p(x) = 0$ and as $I_{p=0}(x) = 0$ if $p(x) \neq 0$. $\square$

Counterexample 17 ($p = 0$ equational invariant $\not\Rightarrow$ p invariant function). We have $\vDash x = 0 \to [x' = x]x = 0$ but $\nvDash x = 1 \to [x' = x]x = 1$, hence $p = 0$ is an equational invariant of $x' = x$ but p is no invariant function, because $p = 1$ is no equational invariant. In particular, we can tell by simulation, whether a polynomial p can be an invariant function, which gives a good falsification test.

The structure of invariant functions is that they form an algebra.

Lemma 18 (Structure of invariant functions). *The invariant functions (or the invariant polynomials) of $x' = \theta \,\&\, H$ form an $\mathbb{R}$-algebra.*

Proof. As a function of time t, let $x(t)$ be a solution of the differential equation under consideration. If p, q are invariant functions and $\lambda \in \mathbb{R}$ is a number (or constant symbol), then $p + q, pq, \lambda p$ are invariant functions, because, for any operator $\oplus \in \{+, \cdot\}$:
$(p \oplus q)(x(t)) = p(x(t)) \oplus q(x(t()) = p(x(0)) \oplus q(x(0)) = (p \oplus q)(x(0))$ $\square$

According to Lemma 18, it is enough to find a generating system of the algebra of invariant functions, because all algebraic expressions built from this generating set are invariant functions. A *generating system* of an algebra is a set S such that the set of all elements that can be formed from S by operations of the algebra coincides with the full algebra. More precisely, the smallest algebra containing S is the full algebra of invariant functions. This generating system is not necessarily small, however, because, whenever p is an invariant function and F an arbitrary (sufficiently smooth) function, e.g., polynomial, then $F(p)$ is an invariant function. This holds accordingly for (sufficiently smooth) functions F with multiple arguments. The situation improves if we take a *functional generating set* G. That is, a set G that gives all invariant functions when closing it under composition with any (sufficiently smooth) function F, i.e., $F(p_1, \dots, p_n)$ is in the closure for all p_i in the closure.

A useful structure of the invariant equations is that they form an ideal. For a fixed dynamics $x' = \theta$ or $x' = \theta \,\&\, H$ we define the following sets of valid formulas and provable formulas, respectively:

$$\begin{aligned}
\mathcal{I}_{=}(\Gamma) &:= \{p \in \mathbb{R}[\boldsymbol{x}] \;:\; \vDash \Gamma \to [x' = \theta \,\&\, H]p = 0\} \\
\mathcal{DCI}_{=}(\Gamma) &:= \{p \in \mathbb{R}[\boldsymbol{x}] \;:\; \vdash_{\mathrm{DI}_{=}+\mathrm{DC}} \Gamma \to [x' = \theta \,\&\, H]p = 0\} \\
r\mathcal{I}_{=} &:= \{p \in \mathbb{R}[\boldsymbol{x}] \;:\; \vDash p = 0 \to [x' = \theta \,\&\, H]p = 0\} \\
r\mathcal{DCI}_{=} &:= \{p \in \mathbb{R}[\boldsymbol{x}] \;:\; \vdash_{\mathrm{DI}_{=}+\mathrm{DC}} p = 0 \to [x' = \theta \,\&\, H]p = 0\}
\end{aligned}$$

The set $\mathcal{I}_{=}(\Gamma)$ collects the polynomials whose variety forms an invariant equation ($p \in \mathcal{I}_{=}(\Gamma)$). The set $\mathcal{DCI}_{=}(\Gamma)$ collects the polynomials for whose zero set it is provable using equational differential invariants ($DI_{=}$) and differential cuts (DC) that they are invariant equations ($p \in \mathcal{DCI}_{=}(\Gamma)$). The sets $\mathcal{I}_{=}(\Gamma)$ and $\mathcal{DCI}_{=}(\Gamma)$ are relative to a d$\mathcal{L}$ formula (or set) Γ that is used as assumption.

The reflexive sets $r\mathcal{I}_=$ and $r\mathcal{DCI}_=$, instead, assume that the precondition and postcondition are identical. It turns out that the reflexive versions do not have a very well-behaved structure (see the following proof). The invariant sets $\mathcal{I}_=(\Gamma)$ and $\mathcal{DCI}_=(\Gamma)$, instead, are well-behaved and form a chain of differential ideals.

Lemma 19 (Structure of invariant equations). *Let Γ be a set of* d$\mathcal{L}$ *formulas, then $\mathcal{DCI}_=(\Gamma) \subseteq \mathcal{I}_=(\Gamma)$ is a chain of differential ideals (with respect to the derivation $\theta \cdot \nabla$, in particular $(\theta \cdot \nabla)p \in \mathcal{DCI}_=(\Gamma)$ for all $p \in \mathcal{DCI}_=(\Gamma)$). Furthermore, the varieties of these ideals are generated by a single polynomial.*

Proof. We prove each of the stated properties.

1. The inclusion follows from soundness. The inclusion $r\mathcal{DCI}_= \subsetneq r\mathcal{I}_=$ even still holds for $r\mathcal{I}_=$.
2. It is easy to see that $p, q \in \mathcal{I}_=(\Gamma)$ and $r \in \mathbb{R}[\boldsymbol{x}]$ imply $p + q, rp \in \mathcal{I}_=(\Gamma)$. Both properties do not hold for r$\mathcal{I}_=$, because $x, x^2 \in r\mathcal{I}_=$ for the dynamics $x' = x$, but the sum/product $x^2 + x = x(x + 1) \notin r\mathcal{I}_=$
3. Let $p, q \in \mathcal{DCI}_=(\Gamma)$, then $p + q \in \mathcal{DCI}_=(\Gamma)$, because $\Gamma \to p = 0 \land q = 0$ implies $\Gamma \to p + q = 0$ (for the antecedent) and $\theta \cdot \nabla$ is a linear operator:
$$(\theta \cdot \nabla)(p + q) = (\theta \cdot \nabla)p + (\theta \cdot \nabla)q = 0 + 0 = 0$$
The second equation holds after sufficiently many uses of DC that are needed to show that $p, q \in \mathcal{DCI}_=(\Gamma)$.
4. Let $p \in \mathcal{DCI}_=(\Gamma)$ and $r \in \mathbb{R}[\boldsymbol{x}]$, then $rp \in \mathcal{DCI}_=(\Gamma)$, because $\Gamma \to p = 0$ implies $\Gamma \to rp = 0$ (for the antecedent) and $\theta \cdot \nabla$ is a derivation operator:
$$(\theta \cdot \nabla)(rp) = p(\theta \cdot \nabla)r + r\underbrace{(\theta \cdot \nabla)p}_{0} = \underbrace{p}_{0}(\theta \cdot \nabla)r = 0$$
The second equation holds after sufficiently many uses of DC that are needed to show that $p \in \mathcal{DCI}_=(\Gamma)$. The last equation holds after one more use of DC by $p = 0$, which entails $p = 0$ on the (new) domain of evolution $H \land p = 0$.
5. The fact that the ideal $\mathcal{DCI}_=(\Gamma)$ is a differential ideal follows from [20, Lem 3.7], which just uses differential weakening. In detail: $p \in \mathcal{DCI}_=(\Gamma)$ implies that $(\theta \cdot \nabla)p = 0$ is provable after sufficiently many DC. Hence, after the same DC, invariance of $(\theta \cdot \nabla)p = 0$ is provable by DW.
6. From $p \in \mathcal{I}_=(\Gamma)$, we conclude $(\theta \cdot \nabla)p \in \mathcal{I}_=(\Gamma)$ as follows. Let $p(x(t)) = 0\ \forall t$. Then $((\theta \cdot \nabla)p)(x(t)) = \sum_i \theta_i(x(t))\frac{\partial p}{\partial x_i}(x(t)) = 0$ follows from the necessity direction of Theorem 3.
7. $p = 0 \land q = 0$ is a propositional equation that is invariant iff $p^2 + q^2 \in \mathcal{I}(\Gamma)$, i.e., $p^2 + q^2 = 0$ gives an invariant equation. The same holds for $\mathcal{DCI}_=(\Gamma)$ by previous work [19,23]. By repeating this construction, we obtain a variety generated by a single polynomial, because, by Hilbert's basis theorem [12], every ideal in the (multivariate) polynomial ring of a Noetherian ring (e.g., a field) is finitely generated ideal. Yet the ring of polynomials is not a principal ideal domain except in dimension 1.

8. $p = 0 \lor q = 0$ is a propositional equational invariant iff $pq \in \mathcal{I}_=(\Gamma)$, i.e., $pq = 0$ gives an invariant equation. The same holds for $\mathcal{DCI}_=(\Gamma)$ by previous work [19,23]. $\square$

Observe that the differential cut rule DC needs to be included to make $\mathcal{DCI}_=$ an ideal (not closed under multiplication with other polymials). Without differential cuts, the set of provable equational differential invariants is generally no ideal. As a corollary to Lemma 19, it is sufficient to find a complete set of differential ideal generators, because these generators describe all other invariants. Without taking functional generators into account, there are still infinitely many invariant equations, because every invariant function induces infinitely many invariant equations by Lemma 16.

According to Lemma 19, however, there is a *single generator of the variety of the differential ideals*, which is the most informative invariant.

5 Assuming Equations and Equational Invariants

Theorem 3 gives an equivalence characterization of invariant functions on open domains. Another seminal result due to Lie provides a similar equivalence characterization for invariant equations of full rank. This equivalence characterization assumes the invariant F during its proof, which is not sound in general; see Counterexample 8. In the case of full rank, this is different.

Theorem 20 (Lie [15,16][17, Theorem 2.8]). *The following rule is sound*

$$(\overleftarrow{DI}_p)\ \frac{\bigwedge_{i=1}^{n} p_i = 0 \to [x' = \theta \,\&\, H] \bigwedge_{i=1}^{n} p_i = 0}{H \land \bigwedge_{i=1}^{n} p_i = 0 \to \bigwedge_{i=1}^{n} (\theta \cdot \nabla) p_i = 0}$$

If rank $\frac{\partial p_i}{\partial x_j} = n$ *on* $H \land \bigwedge_{i=1}^{n} p_i = 0$*, then the premise and conclusion are equivalent.*

Rule $\overleftarrow{DI}_p$ provides a necessary condition for an equation system to be an invariant and can, thus, be used to disprove invariance. Rule $DI_=$ provides a sufficient condition, but implies a stronger property (invariant function instead of just invariant equation). In the full rank case, $\overleftarrow{DI}_p$ is an equivalence and can decide whether $\bigwedge_{i=1}^{n} p_i = 0$ is an invariant equation. Whether $\bigwedge_{i=1}^{n} p_i = 0$ satisfies the full rank condition is decidable in real-closed fields, but nontrivial without optimizations. The invariant in Example 9 has full rank 2, except when $x = y = 0$, which does not satisfy the invariant $x^2 + y^2 = 1$:

$$\begin{pmatrix} \frac{\partial(x^2+y^2-1)}{\partial x} & \frac{\partial(x^2+y^2-1)}{\partial y} & \frac{\partial(x^2+y^2-1)}{\partial e} \\ \frac{\partial(e-x)}{\partial x} & \frac{\partial(e-x)}{\partial y} & \frac{\partial(e-x)}{\partial e} \end{pmatrix} = \begin{pmatrix} 2x & 2y & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

In Counterexample 8, however, the full rank condition is, in fact, violated, since $\frac{\partial(x^2-6x+9)}{\partial y} = 0$ and $\frac{\partial(x^2-6x+9)}{\partial x} = 2x-6$ has a zero when $x=3$, which satisfies $x^2-6x+9=0$. This explains why it was not sound to assume $x^2-6x+9=0$ when attempting to prove it.

It is sound to use equations in the following style (also see [30]):

Theorem 21. *This rule is sound for any choice of smooth functions $Q_{i,j}$:*

$$(\overrightarrow{DI}_p)\ \frac{H \rightarrow \bigwedge_{i=1}^{n} (\theta \cdot \nabla)p_i = \sum_j Q_{i,j}p_j}{\bigwedge_{i=1}^{n} p_i = 0 \rightarrow [x' = \theta \,\&\, H] \bigwedge_{i=1}^{n} p_i = 0}$$

If rank $\frac{\partial p_i}{\partial x_j} = n$ *on* $H \wedge \bigwedge_{i=1}^{n} p_i = 0$*, then the premise of* $\overrightarrow{DI}_p$ *is equivalent to the conclusion of* $\overleftarrow{DI}_p$.

Proof. This result follows from [17], since the premise of $\overrightarrow{DI}_p$ is equivalent to the conclusion of $\overleftarrow{DI}_p$ by [17, Proposition 2.10] in the maximal rank case. We only sketch the (simple) soundness direction for $n=1$ and $H \equiv$ *true*. At any ζ, by Lemma 2, the premise, and the antecedent of the conclusion:

$$\frac{\mathrm{d}\varphi(t)[\![p]\!]}{\mathrm{d}t}(\zeta) = \varphi(\zeta)[\![(\theta \cdot \nabla)p]\!] = \varphi(\zeta)[\![Qp]\!] = \varphi(\zeta)[\![Q]\!] \cdot \varphi(\zeta)[\![p]\!]$$
$$\varphi(0)[\![p]\!] = 0$$

The constant function zero solves this linear differential equation (system). Since solutions are unique (Q and p smooth), this implies $\varphi(\zeta)[\![p]\!] = 0$ for all ζ. □

According to Theorem 20, it is necessary for invariance of $\bigwedge_{i=1}^{n} p_i = 0$ that $(\theta \cdot \nabla)p_i$ is in the variety, i.e., $(\theta \cdot \nabla)p_i \in V(p_1, \dots, p_n)$ for all i. But, according to Theorem 21 it is only sufficient if $(\theta \cdot \nabla)p_i$ is in the ideal $(p_1, \dots, p_n)$ generated by the p_j, i.e., the set $\{\sum_j Q_j p_j : Q_j \in \mathbb{R}[x]\}$. In the full rank case, both conditions are equivalent.

Counterexample 22 (Full rank). Full rank is required for equivalence. For example, $h := x-1$ vanishes on $p := (x-1)^2 = 0$, but no smooth function Q satisfies $h = Qp$, since the required $Q := (x-1)^{-1}$ has a singularity at $p=0$.

6 Partial Differential Equations and the Inverse Characteristic Method

In this section, we study the connection of differential invariants with partial differential equations. The operator $\theta \cdot \nabla$ defined in (1) is a differential operator.

Then the premise $H \to (\theta \cdot \nabla)p$ of DI_c, which is the same as the premise of $DI_=$, is a partial differential equation on the domain H.

$$(\theta \cdot \nabla)p = 0 \quad \text{on } H \tag{5}$$

This equation is a first-order, linear, homogeneous partial differential equation, which are well-behaved partial differential equations. By Theorem 3, p is a solution of the partial differential equation (5) on domain H iff p is an invariant function of $x' = \theta \,\&\, H$. Thus, with the caveats explained in Section 3, solving partial differential equations gives a complete approach to generating invariant functions, which are the strongest type of differential invariants.

This approach first seems to be at odds with what we wanted to achieve in the first place. Differential equations are complicated, their solutions hard to compute. So we work with differential invariants instead, which are perfect for verification if only we find them. In order to find differential invariants, we solve a partial differential equation, which, in general, is even harder than solving ordinary differential equations. In fact, many numerical and symbolic algorithms for solving partial differential equations are based on solving a number of ordinary differential equation systems as subproblems. The *characteristic method*, see [5, Theorem 3.2.1][32, §1.13.1.1], studies the characteristic ordinary differential equations belonging to a partial differential equation in order to understand the partial differential equation.

We nevertheless proceed this way and call it the *inverse characteristic method*, i.e., the study of properties of ordinary differential equations by studying the partial differential equation belonging to its Lie-type differential operator.

Theorem 23 (Inverse characteristic method). *A (sufficiently smooth) function f is an invariant function of the differential equation $x' = \theta$ on the domain H iff f solves the partial differential equation (5) on H, i.e.,*

$$(\theta \cdot \nabla)f = 0 \quad \textit{on } H$$

Proof. This is a consequence of Theorem 3. □

The inverse characteristic method is insightful for two reasons. First, it identifies a mathematically well-understood characterization of the problem of generating differential invariants, at least for the equational case of invariant functions on domains. Second, the inverse characteristic method can be quite useful in practice, because the resulting partial differential equations are rather well-behaved, and solvers for partial differential equations are built on very mature foundations. Note that it is beneficial for the purposes of building a verification tool that the partial differential equation solver can work as an oracle and does not need to be part of the trusted computing base, since we can easily check its (symbolic) solutions for invariance by rule DI_c just using symbolic derivatives and polynomial algebra.

Example 24 (Deconstructed aircraft). For the deconstructed aircraft from Counterexample 5, the dynamics yields the corresponding partial differential equation

$$-y\frac{\partial f}{\partial x} + e\frac{\partial f}{\partial y} - y\frac{\partial f}{\partial e} = 0$$

whose solution can easily be computed to be

$$f(x,y,e) = g\left(e - x, \frac{1}{2}(2ex - x^2 + y^2)\right)$$

Thus, the solution is a function g of $e - x$ and of $\frac{1}{2}(2ex - x^2 + y^2)$, which turns both terms into invariant functions:

$$e - x \tag{6}$$

$$2ex - x^2 + y^2 \tag{7}$$

Contrast this with the invariant equation $(e^2 + y^2 - 1)^2 + (e - x)^2 = 0$ from the proof of (4) in Example 11. In order to relate this creative invariant to the systematically constructed invariants (6)–(7), we note that the initial state and postcondition in (4) is $x^2 + y^2 = 1 \wedge e = x$. Hence, $y^2 = 1 - x^2, e = x$, which we substitute in (7) to obtain $2xx - x^2 + (1 - x^2) = 1$. Thus, for the purpose of proving (4), the initial value for (6) is 0 and that for (7) is 1. Using $e - x = 0$, the invariant $e^2 + y^2 - 1$ can be obtained from (7) and the initial value 1 by polynomial reduction.

Example 25 (Aircraft). For the actual aircraft dynamics in Example 12, the corresponding partial differential equation

$$d_1\frac{\partial f}{\partial x_1} + d_2\frac{\partial f}{\partial x_2} - \omega d_2\frac{\partial f}{\partial d_1} + \omega d_1\frac{\partial f}{\partial d_2} = 0$$

whose solution can easily be computed to be (recall $\omega \neq 0$)

$$f(x_1,x_2,d_1,d_2) = g\left(d_2 - \omega x_1, \frac{d_1 + \omega x_2}{\omega}, \frac{1}{2}(d_1^2 + 2\omega d_2 x_1 - \omega^2 x_1^2)\right)$$

revealing the invariant functions $d_2 - \omega x_1, d_1 + \omega x_2, d_1^2 + 2\omega d_2 x_1 - \omega^2 x_1^2$. From these, the creative invariant in Example 12 can be constructed in retrospect with initial value 0, 0, and $\omega^2 p^2$, respectively. The value $\omega^2 p^2$ can be found either by polynomial reduction or by substituting $\omega x_1 = d_2$ in as follows

$$d_1^2 + 2\omega d_2 x_1 - \omega^2 x_1^2 = d_1^2 + 2d_2^2 - d_2^2 = d_1^2 + d_2^2 = \omega^2 p^2$$

7 Conclusions and Future Work

Differential invariants are a natural induction principle for differential equations. The structure of general differential invariants has been studied previously. Here, we took a differential operator view and have studied the case of equational differential invariants in more detail. We have related equational differential invariants to Lie's seminal work and subsequent results about Lie groups. We have shown how the resulting equivalence characterization of invariant equations on open domains can be used, carefully illustrate surprising challenges in invariant generation, explain why they exist, and show with which techniques they can be overcome. We have studied the structure of invariant functions and invariant equations, their relation, and have shown that, in the presence of differential cuts, the invariant equations and provable invariant equations form a chain of differential ideals and that their varieties are generated by a single invariant. Finally, we relate differential invariants to partial differential equations and explain how the inverse characteristic method reduces the problem of equational differential invariant generation to that of solving partial differential equations.

The results we present in this paper relate equational differential invariants to other problems. They show equivalence characterizations and methods for generating equational differential invariants. While the connection with other aspects of mathematics makes a number of classical results available, their complexity indicates that the study of differential invariants has the potential for many further discoveries. In this paper, we have focused exclusively on the equational case. In the theory of differential invariants, however, the equational and general case have quite different characteristics [23]. The general case of differential invariants that are logical formulas with equations and inequalities has been studied elsewhere [23].

Acknowledgements. I thank David Henriques and the PC chairs of ITP for helpful feedback.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. 138(1), 3–34 (1995)
2. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. 12(3), 299–328 (1991)
3. Cox, D.A., Little, J., O'Shea, D.: Ideals, Varieties and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer (1992)
4. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. 5(1/2), 29–35 (1988)
5. Evans, L.C.: Partial Differential Equations. Graduate Studies in Mathematics, 2nd edn., vol. 19. AMS (2010)
6. Gentzen, G.: Untersuchungen über das logische Schließen. II. Math. Zeit. 39(3), 405–431 (1935)
7. Grigor'ev, D.Y.: Complexity of Quantifier Elimination in the Theory of Ordinary Differential Equations. In: Davenport, J.H. (ed.) ISSAC 1987 and EUROCAL 1987. LNCS, vol. 378, pp. 11–25. Springer, Heidelberg (1989)
8. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, Malik [9], pp. 190–203
9. Gupta, A., Malik, S. (eds.): CAV 2008. LNCS, vol. 5123. Springer, Heidelberg (2008)
10. Hartshorne, R.: Algebraic Geometry. Graduate Texts in Mathematics, vol. 52. Springer (1977)

11. Henzinger, T.A.: The theory of hybrid automata. In: LICS, pp. 278–292. IEEE Computer Society, Los Alamitos (1996)
12. Hilbert, D.: Über die Theorie der algebraischen Formen. Math. Ann. 36(4), 473–534 (1890)
13. Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia. IEEE Computer Society (2012)
14. Lie, S.: Über Differentialinvarianten, vol. 6. Teubner (1884); English Translation: Mr. Ackerman (1975); Sophus Lie's 1884 Differential Invariant Paper. Math. Sci. Press, Brookline, Mass.: (1884)
15. Lie, S.: Vorlesungen über continuierliche Gruppen mit geometrischen und anderen Anwendungen. Teubner, Leipzig (1893)
16. Lie, S.: Über Integralinvarianten und ihre Verwertung für die Theorie der Differentialgleichungen. Leipz. Berichte 49, 369–410 (1897)
17. Olver, P.J.: Applications of Lie Groups to Differential Equations, 2nd edn. Springer (1993)
18. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. 41(2), 143–189 (2008)
19. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
20. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
21. Platzer, A.: The complete proof theory of hybrid systems. In: LICS [13]
22. Platzer, A.: Logics of dynamical systems (invited tutorial). In: LICS [13]
23. Platzer, A.: The structure of differential invariants and differential cut elimination. In: Logical Methods in Computer Science (to appear, 2012)
24. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, Malik [9], pp. 176–189
25. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. CAV 2008 35(1), 98–120 (2009); Special issue for selected papers from CAV 2008
26. Platzer, A., Clarke, E.M.: Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009)
27. Platzer, A., Quesel, J.-D., Rümmer, P.: Real World Verification. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 485–501. Springer, Heidelberg (2009)
28. Prajna, S., Jadbabaie, A.: Safety Verification of Hybrid Systems Using Barrier Certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
29. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE T. Automat. Contr. 52(8), 1415–1429 (2007)
30. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. Form. Methods Syst. Des. 32(1), 25–55 (2008)
31. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1951)
32. Zeidler, E. (ed.): Teubner-Taschenbuch der Mathematik. Teubner (2003)

Abella: A Tutorial

Andrew Gacek

Rockwell Collins, Inc.

Abella is an interactive theorem prover aimed at developing the meta-theory of programming languages, logics, and other systems with binding. Abella uses a higher-order abstract syntax representation of object systems, and the Abella logic provides rich features for manipulating this representation. The result is that binding-related issues are handled automatically by Abella, so users are freed to focus directly on the interesting parts of their proofs. This tutorial explains these concepts in depth and illustrates them through examples from various domains.

The most fundamental component of Abella is higher-order abstract syntax which enriches traditional abstract syntax with λ-binders. These terms succinctly encode object languages with binding structure such as programming languages and logics. Moreover, common operations such as renaming of bound variables and capture-avoiding substitution correspond exactly to the rules of λ-conversion which are integrated into the term language. Thus these issues are lifted up into the logic of Abella where they are treated once-and-for-all, freeing users from tedious and error-prone work related to binding.

The reasoning logic underlying Abella is a first-order logic with a higher-order term structure [2]. The logic is enriched with user-defined inductive and co-inductive definitions with corresponding induction and co-induction rules. In order to reason over higher-order terms, two recent research advancements are incorporated into the logic. The first is the ∇-quantifier which represents *generic* quantification. Intuitively, this quantifier introduces a "fresh name" which is used to deconstruct binding structure in a logically sensible way. The second research advancement is the notion of *nominal abstraction* which allows for sophisticated recognition of terms which have resulted from uses of the ∇-quantifier. These additions to the logic allow additional binding-related issues to be treated once-and-for-all by Abella.

Abella includes a specification logic which can optionally be used to encode object language specifications. The specification logic is executable; in fact, it is a subset of the λProlog language. This allows for rapid prototyping and checking of specifications. The deeper benefit of the specification logic is that it has been encoded into the reasoning logic and general properties of it have been proven. These properties typically correspond to useful lemmas about the structure of object language judgments. For example, the cut-elimination result on the specification logic implies a substitution principle on typing judgments encoded in the specification logic. This is called the *two-level logic approach* to reasoning, and in practice it yields significant benefits.

Abella has been used to prove a variety of results [1]. In the domain of logic, it has been used to prove cut-elimination for the LJ sequent calculus, correctness and completeness for a focusing logic, and equivalences between natural deduction, Hilbert calculus, and sequent calculus. For programming languages, it has been used to solve the POPLmark Challenge, to prove the equivalence of various notions of evaluation, and to assist graduates students in an advanced programming languages course. In the λ-calculus, it has been used to prove the Church-Rosser property, standardization, and strong normalization (in the typed setting). Most recently, Abella was used to prove properties of various notions of bisimulation in the π-calculus. These developments and more including downloads, tutorials, and papers are available at http://abella.cs.umn.edu/.

References

1. Gacek, A.: The Abella Interactive Theorem Prover (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 154–161. Springer, Heidelberg (2008)
2. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. Information and Computation 209(1), 48–73 (2011)

A Cantor Trio: Denumerability, the Reals, and the Real Algebraic Numbers

Ruben Gamboa and John Cowles

University of Wyoming, Laramie, WY, USA
{ruben,cowles}@uwyo.edu
http://www.cs.uwyo.edu/~ruben

Abstract. We present a formalization in ACL2(r) of three proofs originally done by Cantor. The first two are different proofs of the non-denumerability of the reals. The first, which was described by Cantor in 1874, relies on the completeness of the real numbers, in the form that any infinite chain of closed, bounded intervals has a non-empty intersection. The second proof uses Cantor's celebrated diagonalization argument, which did not appear until 1891. The third proof is of the existence of real transcendental (i.e., non-algebraic) numbers. It also appeared in Cantor's 1874 paper, as a corollary to the non-denumerability of the reals. What Cantor ingeniously showed is that the algebraic numbers are denumerable, so every open interval must contain at least one transcendental number.

Keywords: ACL2, nonstandard analysis, non-denumerability of the reals, denumerability of algebraic numbers.

1 Introduction

In an important paper first published in 1874[1,2] and later popularized by Dunham among others [3], Cantor presented a new proof of the existence of transcendental numbers, those numbers that are not the root of any polynomial with rational coefficients. Cantor's proof was quite unlike Liouville's earlier demonstration of a transcendental number. While Liouville actually constructed a transcendental number (indeed, a whole family of them), Cantor used a counting argument to show that the set of real numbers must include many transcendental numbers.

Cantor's counting argument proceeds as follows. First, he showed that no sequence $\{x_1, x_2, \dots\}$ of real numbers can completely enumerate all the numbers in an open interval (a, b). That is, there must be some $x \in (a, b)$ such that $x \notin \{x_1, x_2, \dots\}$. Second, he constructed an enumeration of the algebraic numbers, that is, all the roots of all polynomials with rational coefficients. Since the algebraic numbers could be placed in a sequence, it followed that every non-empty open interval must contain at least one number that is not among the algebraic numbers, i.e., a transcendental number.

Although Cantor's 1874 paper emphasized the application to transcendental numbers, the more revolutionary result was the non-denumerability of the real numbers! In 1891, Cantor proved, by diagonalization, that non-denumerable sets exist. This diagonalization proof is easily adapted to showing that the reals are non-denumerable, which is the proof commonly presented today [4,2].

We present a formalization of Cantor's two proofs of the non-denumerability of the reals in ACL2(r). In addition, we present a formalization of Cantor's application of this theorem to the existence of transcendental numbers. The formalizations rely on some uncommon techniques of ALC2(r). So we begin the presentation in Sect. 2 by briefly describing ACL2(r) and the main techniques on which the proofs rely. We follow in Sect. 3 with the formalization of Cantor's two proofs of the non-denumerability of the reals. Then in Sect. 4 we present Cantor's enumeration of the algebraic numbers, which immediately establishes the existence of an infinite number of transcendental numbers. Finally, we present some concluding remarks in Sect. 5.

2 Background: ACL2(r)

In this section, we briefly describe ACL2(r), a variant of ACL2 with support for the real numbers, including an overview of nonstandard analysis, the foundational theory of ACL2(r). Our description is limited to those elements of ACL2 and ACL2(r) that are needed for the main results described later in this paper. Readers familiar with ACL2 or ACL2(r) may wish to skip this section.

In the tradition of the Boyer-Moore family of theorem provers, ACL2 is a first-order theorem prover with a syntax similar to Common Lisp's [5]. The primary inference rules are term rewriting with equality (and with respect to other equivalence relations) and induction up to ϵ_0. ACL2 supports the explicit introduction of new function symbols via explicit definitions as well as implicitly via constraints, using the events[1] `defun` and `encapsulate`, respectively. In addition, ACL2 permits the introduction of "choice" functions via Skolem axioms using the `defchoose` event. For example, let $\phi(x, y)$ be a formula whose only free variables are x and y. Then the Skolem axiom introducing the function f from the formula $\phi(x, y)$ with respect to y is

$$(\forall x, y)(\phi(x, y) \Rightarrow \phi(x, f(x)))$$

What this axiom states is that the new function f can "choose" an appropriate y for a given x as long as such a y exists. For example, if $\phi(x, y)$ states that $y^2 = x$, then the choice function will select one of $\pm\sqrt{x}$ for a non-negative real x. What it does for other values of x is unspecified.

Choice functions in ACL2 are also used to define expressions that capture the essence of existential and universal quantifiers in the event `defun-sk`. For example, the following event captures the concept that an object has a square root:

[1] An ACL2 "event" is the unit of interaction between the user and the theorem prover, e.g., a command.

```
(defun-sk exists-square-root (x)
 (exists (y)
   (and (realp y)
        (equal (* y y) x))))
```

These "quantification" functions can then be used in theorems, such as the following, which states that non-negative reals have square roots:

```
(defthm nonneg-reals-have-square-roots
 (implies (and (realp x) (<= 0 x))
          (exists-square-root x)))
```

Choice functions in ACL2 are also used to justify the definition of "partial" functions with the event `defpun` [6]. The basic idea behind `defpun` is that under certain circumstances a recursive expression can be used to define a function symbol, even when there is no guarantee that the function terminates for all inputs. For example, consider the following function, which returns the next highest prime number.

```
(defpun next-prime (n)
 (if (primep (1+ n))
     (1+ n)
   (next-prime (1+ n))))
```

A naive attempt to define `next-prime` by replacing `defpun` with `defun` will fail, because ACL2 is unable to find a measure that decreases in the recursive call. However, using `defpun`, the definition is admitted. The function `next-prime` defined in this way is not truly partial, because it has a value for every input. Rather, it is underspecified, because its value is only known for certain input values. E.g., it would be possible to prove that `(next-prime 80)` is equal to 83, but it would not be possible to prove what `(next-prime 1/2)` is equal to, even though it must surely be equal to *something*, since ACL2 is a logic of total functions.

ACL2(r) modifies the base ACL2 theorem prover by introducing notions from nonstandard analysis, as axiomatized by Nelson [7,8]. In Nelson's formulation of nonstandard analysis, the real numbers can be further characterized as standard, small, limited, or large. The standard reals include all the real numbers that can be uniquely characterized, such as 0, 1, π, $\sqrt{2}$, etc. Small reals, also called infinitesimals, are those that are smaller in magnitude than any non-zero standard real. Zero is the only standard number that is also small, but there are other small numbers. Necessarily, there are also large numbers, namely those that are larger in magnitude than all standard reals. Real numbers x that are not large are called limited, and they can always be written as $x = {}^*x + \epsilon$, where ${}^*x$ is standard and ϵ is small. The number ${}^*x$ is called the standard part of x. Finally, two numbers are said to be close if their difference is small. It is important to note that all the usual algebraic properties of the real numbers are still true in nonstandard analysis. E.g., $x \cdot 1/x = 1$ for all non-zero x, whether x is small, large, limited, standard, etc. Also, the properties close, small, and so on have nice algebraic properties. E.g., the sum of two small numbers is small, the product of a small and a limited number is small, and the standard part of the sum of two numbers is the sum of their standard parts.

Formulas and functions in nonstandard analysis are said to be classical if they do not mention any of the "new" functions of nonstandard analysis, i.e., standard, large, etc. Thus, all functions of traditional analysis, such as square root and sine, are classical. One of the most important principles of nonstandard analysis is the transfer principle, which states that any first-order classical formula that is true of all standard values must also be true of all values. That is, in order to prove that a classical formula $P(x)$ is true of all x, it is sufficient to prove that $\mathit{standard}(x) \Rightarrow P(x)$. This principle is captured in the ACL2(r) event `defthm-std`. This same principle also justifies an indirect definitional principle, where only the values of $f(x)$ for standard values of x are specified. Under certain conditions, this is sufficient to define f as the only classical function that maps x to $f(x)$ for all standard values of x.

We conclude this section by mentioning that many of the traditional notions of analysis can be stated more naturally in nonstandard analysis. For example, we say that a sequence of real numbers $\{a_1, a_2, \dots\}$ converges to a value A if and only if a_N is close to A for all large values of N. This is remarkably simpler than the traditional "epsilon" definition of convergence.

3 Non-denumerability of the Reals

In this section, we present two proofs of the non-denumerability of the reals. We start with Cantor's 1874 proof, based on the completeness of the real number line [1]. Then we formalize the more familiar proof based on his 1891 paper [4]. In both cases, we present first a mathematical description of the proof that we actually formalized in ACL2(r), and then we present highlights from the formalization.

Note that this is not the first formalization of the non-denumerability of the reals! As of this writing, there are five others in Freek Wiedijk's list, Formalizing 100 Theorems [9]—and this list is not comprehensive. However, our interest here is not just to prove that the reals are non-denumerable, but to formalize *Cantor's actual arguments* in ACL2(r).

3.1 The First Proof: Using the Completeness of the Reals

3.1.1 The Informal Argument

Consider a sequence $\{s_n\}$ of real numbers and a real interval (a, b). Cantor showed that there must be at least one $x \in (a, b)$ such that $x \notin \{s_n\}$. The following argument is a slight variant of Cantor's argument, which we formalized in ACL2(r).

First, construct an infinite chain of nested, closed, and bounded intervals as follows. Let $[a_0, b_0] = [a, b]$. The interval $[a_1, b_1] \subsetneq [a_0, b_0]$ is then defined as $[s_{i_1}, s_{j_1}]$, where i_1 and j_1 are chosen to be the smallest indexes such that $i_1 < j_1$

and $s_{i_1} < s_{j_1}$ are both in (a_0, b_0)—as long as such indexes can be found. Repeat this process, so that i_n and j_n as the smallest indexes such that $j_{n-1} < i_n < j_n$ and $s_{i_n} < s_{j_n}$ are both in (a_{n-1}, b_{n-1})[2]. By construction, if appropriate indexes can be found at every step, the intervals $[a_n, b_n]$ form an infinite chain of nested, closed, and bounded intervals.

However, the construction can fail at a step if no i_n and j_n can be found such that $j_{n-1} < i_n < j_n$ and s_{i_n} and s_{j_n} are both in (a_{n-1}, b_{n-1}). But if this is the case, we have found a point in $[a_{n-1}, b_{n-1}] \subset (a, b)$ that cannot be one of the $\{s_n\}$, as desired. This is because either (i) for all $i_n > j_{n-1}$, $s_{i_n} \notin (a_{n-1}, b_{n-1})$ or (ii) if i_n is the first index such that $i_n > j_{n-1}$ and $s_{i_n} \in (a_{n-1}, b_{n-1})$, then for all $j_n > i_n$ if $s_{j_n} \in (a_{n-1}, b_{n-1})$, then $s_{j_n} \leq s_{i_n}$. In case (ii) holds, then for all $j_n > i_n$, $s_{j_n} \notin (s_{i_n}, b_{n-1})$.

Conversely, suppose the construction does succeed in building an infinite chain of nested intervals. Then there is some point x such that $x \in [a_n, b_n]$ for all n. The claim is that x is not any of the s_n. This follows by considering the possible values of the indexes i_n and j_n. First, it's clear that $i_1 \geq 1$ and $j_n \geq 2$, since these are the first two indexes that can be considered; in general, $i_n \geq 2n-1$ and $j_n \geq 2n$. Second, we observe that for i such that $j_1 < i < i_2$, $s_i \notin (a_1, b_1)$, since i_2 is chosen to be the first i in that interval. Moreover, for i such that $i_1 < i < j_1$, we also know that $s_i \notin (a_1, b_1)$, since j_1 is the first index such that $j_1 > i_1$ and $s_{j_1} \in [a_0, b_0]$. Combining and generalizing these facts, it follows that for i in the range $i_n \leq i < i_{n+1}$, $s_i \notin (a_n, b_n)$. This statement can be used to show, by induction, that for all i in the range $1 \leq i < i_{n+1}$, $s_i \notin (a_n, b_n)$.

Finally, the two observations above can be combined to observe that if $1 \leq i < 2n+1$, $s_i \notin (a_n, b_n)$, since $i_{n+1} \geq 2(n+1)-1$. In particular, $s_n \notin (a_n, b_n)$ for any n. Since $(a_n, b_n) \supsetneq [a_{n+1}, b_{n+1}]$, it follows that $s_n \neq x$, since x was chosen previously such that $x \in [a_n, b_n]$ for all n.

3.1.2 The Completeness of the Reals

Cantor's proof makes use of the fact that the real numbers are complete, in the form that a sequence of nested, closed, bounded intervals has a non-empty intersection. It is necessary, therefore, to formalize this result in ACL2(r). To do so, we introduce the constrained function `nested-interval`, which represents a sequence of closed intervals, i.e., a mapping from each positive integer n to a pair of real numbers a_n and b_n such that $a_n \leq b_n$. In addition, the intervals are constrained to form a nested chain by requiring that $a_m \leq a_n \leq b_n \leq b_m$ whenever $m \leq n$.

We limit ourselves to standard sequences of nested closed and bounded intervals, since the transfer principle of nonstandard analysis permits us to generalize this result to all sequences later. Since the sequence $\{[a_n, b_n]\}$ is standard, it follows that both a_1 and b_1 are standard. Moreover, since the intervals are nested, we find that $a_1 \leq a_n \leq b_n \leq b_1$ for all n. In particular, $|a_n| \leq \max(|a_1|, |b_1|)$, and this implies that a_n must be limited for all values of n.

[2] Cantor's original proof does not require that $i_n < j_n$. Rather, Cantor finds the next two sequence points in the interval, then chooses i_n and j_n so that $s_{i_n} < s_{j_n}$. That is the only difference between his proof and the one formalized in ACL2(r).

Now, let N be an arbitrary large positive integer—the ACL2(r) constant `i-large-integer` serves this purpose. Since a_N is limited, we can define $A \equiv {}^*a_N$. Notice that A is necessarily standard, since it is the standard part of a limited number. Moreover, for all standard n, $n < N$ (since all standard integers are less than all large integers), and since the intervals are nested, it follows that $a_n \leq a_N$. Taking standard parts of both sides, we can conclude that $a_n \leq {}^*a_N = A$, and using the transfer principle we conclude that $a_n \leq A$ for all n (standard or not).

Similarly, notice that $a_n \leq b_n$ for all n, so that ${}^*a_n \leq {}^*b_n$. Taking standard parts of both sides, it follows that $A \leq b_n$ for all standard values of n, and hence for all values n by the transfer principle. What this means is that we have found a real number A such that $a_n \leq A \leq b_n$ for all n; i.e., $A \in [a_n, b_n]$ for all n, and hence the intersection of the intervals $[a_n, b_n]$ is not empty. This result is summarized in the following ACL2(r) theorem:

```
(defthm standard-part-car-interval-in-intersection
 (and (realp (standard-part-car-interval-large))
      (implies (posp n)
               (and (<= (car (nested-interval n))
                        (standard-part-car-interval-large))
                    (<= (standard-part-car-interval-large)
                        (cdr (nested-interval n))))))
 :hints ...)
```

This argument depends crucially on the use of the transfer principle to show that $a_n \leq A = {}^*a_N \leq b_n$ for all n. However, the transfer principle only applies to classical statements, which this statement is not, since it uses the function standard part. The reason we can do this is that we can define two versions of A, one using `defun` and the other `defun-std`.

```
(defun standard-part-car-interval-large ()
 (standard-part (car (nested-interval (i-large-integer)))))

(defun-std standard-part-car-interval-large-classical ()
 (standard-part-car-interval-large))
```

As explained in the introduction, the version that uses `defun-std` is classical, but its definition is only equal to the expression in the body when the arguments to the function are standard—a condition that is vacuously true in this case, so ACL2(r) can prove that these two definitions are equivalent.

As it turns out, this is the only step in this first proof that uses the non-standard analysis features of ACL2(r). The remainder of the proof could just as easily be carried out in ACL2 (with the exception that it refers to real numbers, not just the rationals).

3.1.3 Constructing the Chain of Nested, Closed, and Bounded Intervals

We now consider some of the highlights of the formalization of the construction of the chain of intervals. The sequence $\{s_n\}$ itself is formalized by defining a constrained function `seq` whose only constraint is that it maps the positive integers to real numbers.

The construction repeatedly looks for the smallest index i such that $i \geq n$ and $s_i \in [a, b]$, for some choice of a, b, and n. This is implemented by the function `next-index-in-range`:

```
(defpun next-index-in-range (n A B)
 (if (in-range (seq n) A B)
     n
   (next-index-in-range (1+ n) A B)))
```

Of course, there is no guarantee that such an i can be found, so the function `next-index-in-range` is not guaranteed to always terminate as written. Thus, it can only be admitted into ACL2(r) by the use of `defpun` instead of `defun`. However, this also means that to reason about `next-index-in-range`, we have to consider the possibility that it fails for a given `n`, `A`, and `B`.

To do so requires the use of existential quantifiers, which we can do with `defun-sk`. The following function, for example, is used to determine which values of `n`, `A`, and `B` lead to success:

```
(defun-sk exists-next-index-in-range (n A B)
 (exists m
         (and (posp m)
              (<= n m)
              (in-range (seq m) A B))))
```

It is then possible to define the function `cantor-sequence-indexes` which returns the nth interval in the construction, or `nil` if no such interval can be found.

Now, suppose that `cantor-sequence-indexes` ever returns `nil`; i.e., that the construction of nested intervals stops after a finite number of iterations. This means that `next-index-in-range` must have been false for some choice of n, A, and B. In this case, we find a point $x \notin \{s_n\}$ as follows. First, we observe that given the choice of n, none of the s_i with $i > n$ can be in $[A, B]$. This means that at most a finite number of points (i.e., n) in the sequence can be in $[A, B]$. But then it is easy to find a point $x \in (A, B)$ that is not one of these n points. The simple, recursive function `counter-example` does just that.

So now suppose that `cantor-sequence-indexes` never returns `nil`; i.e., that the construction of nested intervals continues ad infinitum. It can be easily shown that the resulting sequence of intervals satisfies all the constraints of an infinite chain of nested, closed, bounded intervals, as defined in Sect. 3.1.2. Thus, the theorems of that section apply to `cantor-sequence-indexes`, and we can conclude using the principle of functional instantiation that there is some point that is in each of the intervals.

At this point, the remainder of the proof can be carried out. The only difficult portion is the proof that for i in the range $1 \leq i < i_{n+1}$, $s_i \notin [a_n, b_n]$. This was done using natural induction on n, with the key lemmas being that the theorem holds for i in the range $i_n \leq i < i_{n+1}$, and that the intervals are nested, so that if $s_i \notin [a_{n-1}, b_{n-1}]$, then it trivially follows that $s_i \notin [a_n, b_n]$.

The final statement of the theorem makes use of the (limited) support for quantifiers in ACL2(r). Because this support does not extend directly to nested quantifications, it is necessary to introduce several functions to express the result. First, the function `exists-in-sequence` captures the notion that x is one of the $\{s_n\}$. Similarly, the function `exists-in-interval-but-not-in-sequence` states that x is in the interval $[a, b]$ but is *not* one of the $\{s_n\}$. This uses `exists-in-sequence` to capture that nested quantification. With that, the final statement of the theorem is that `exists-in-interval-but-not-in-sequence` holds (over an arbitrary interval).

```
(defun-sk exists-in-sequence (x)
 (exists i
         (and (posp i)
              (equal (seq i) x))))

(defun-sk exists-in-interval-but-not-in-sequence (A B)
 (exists x
         (and (realp x)
              (< A x)
              (< x B)
              (not (exists-in-sequence x)))))

(defthm reals-are-not-countable
 (exists-in-interval-but-not-in-sequence (a) (b))
 :hints ...)
```

3.2 The Second Proof: Using Diagonalization

3.2.1 The Informal Argument

Cantor's second proof of the non-denumerability of the real numbers is based on diagonilization. The familiar idea is as follows. As before, let $\{s_n\}$ be a sequence of real numbers, but this time further assume that $s_n \in [0, 1]$.

Now, any number x such that $x \in [0, 1]$ can be written as a sequence of digits, e.g., $x = 0.d_1d_2d_3\ldots$, where each digit d_i is an integer from 0 to 9. This expansion of x into digits follows from the fact that x can be written in the form $x = \sum_{i=1}^{\infty} \frac{d_i}{10^i}$.

Obviously, if two numbers have the same expansions they are equal to each other. However, it is possible for two different expansions to result in the the same number, e.g., $0.1999\ldots = 0.2000\ldots$. This strictly technical difficulty prevents us from casually swapping between the number and its representation as a sequence of digits, but this difficulty can be addressed in a number of different ways. We chose to address it by considering how different two expansions have to be in order for them to represent two different numbers.

Suppose $x = \sum_{i=1}^{\infty} \frac{d_i}{10^i}$ and $y = \sum_{i=1}^{\infty} \frac{e_i}{10^i}$, where each of the d_i and e_i are digits, and suppose that k is such that $d_k \neq e_k$. Obviously, we can divide the expansion of x as follows, and similarly for y:

$$x = \sum_{i=1}^{\infty} \frac{d_i}{10^i} = \left(\sum_{i=1}^{k-1} \frac{d_i}{10^i}\right) + \frac{d_k}{10^k} + \left(\sum_{i=k+1}^{\infty} \frac{d_i}{10^i}\right) = L_x + \frac{d_k}{10^k} + R_x,$$

where L_x and R_x (and similarly L_y and R_y) are introduced as shorthands for the respective sums. We can now find bounds for the two sums on the right. For instance, since $d_i \leq 9$ for all i, it follows that $R_x = \sum_{i=k+1}^{\infty} \frac{d_i}{10^i} \leq 1/10^k$. This also gives us a (rough, but sufficient) estimate of the maximum difference between R_x and R_y, i.e., $|R_x - R_y| \leq 2/10^k$. Similarly, we can consider possible differences between L_x and L_y. This time, we find a minimum difference, i.e., $|L_x - L_y| \geq 10/10^k$, unless $L_x = L_y$. This follows, because if L_x and L_y are different, then the minimum difference is when only the least significant digits differ and then only by 1, which yields a minimum difference of $1/10^{k-1}$.

So when $d_k \neq e_k$, we have

$$|x - y| \leq |L_x - L_y| + |d_k - e_k| + |R_x - R_y|.$$

We have an upper bound for $|R_x - R_y|$, so if $|d_k - e_k|$ is large enough, the difference between R_x and R_y will be insufficient to make x and y equal to each other. That's enough to show that if $L_x = L_y$, then $x \neq y$. So suppose $L_x \neq L_y$. Again, we have a lower bound for the difference, so as long as $|d_k - e_k|$ is small enough, the difference will be insufficient to make x and y equal. Thus, as long as d_k is sufficiently different from e_k (i.e., $3 \leq |d_k - e_k| \leq 7$), we can show that $x \neq y$.

ACL2(r) does not support infinite computations, such as $x = \sum_{i=1}^{\infty} \frac{d_i}{10^i}$. Instead, we use the standard part of a partial sum up to a large integer. So, if N is an arbitrary, fixed, large integer, we can say $x = {}^*\sum_{i=1}^{N} \frac{d_i}{10^i}$. As before, we can split this sum into three parts, so that $x = {}^*(L_x + d_k + R_x)$, where L_x is as before and R_x is similar, but with upper limit N instead of ∞. We can limit ourselves to standard x and k, since the transfer principle will carry over the results to all x and k. When x is standard, so are L_x and d_k (as these are finite sums), so $x = L_x + d_k + {}^*R_x$. Earlier, the upper and lower bounds on L_x and R_x were enough to show that if d_k is sufficiently different from e_k, then $x \neq y$. But the argument is more subtle in the nonstandard case: It is not enough to show that the sums $L_x + d_k + R_x$ and $L_y + e_k + R_y$ differ, because two numbers may be different even though their standard parts are the same. So what we need to show is that these two sums have different standard parts, or equivalently that they are not close to each other. We can do so by observing that if d_k is sufficiently different from e_k, then $|(L_x + d_k + R_x) - (L_y + e_k + R_y)| \geq 2/10^k$. Since k is standard, $2/10^k$ is not small. This means that $L_x + d_k + R_x$ and $L_y + e_k + R_y$ must have different standard parts, so $x \neq y$.

We have established that every number $x \in [0,1]$ can be converted into a sequence of digits, and that whenever two sequences of digits are "sufficiently different" at a given position, the numbers that correspond to those sequences are different. That is all we need to carry out Cantor's diagonalization argument.

Start with the sequence $\{s_n\}$ and convert each s_n into a sequence of digits, $s_n = 0.d_{n,1}d_{n,2}d_{n,3}\ldots$. Then construct a new sequence $\{t_n\}$ by choosing t_n to be sufficiently different from $d_{n,n}$—for example, let $t_n = 7$ if $d_{n,n} < 5$, and $t_n = 2$ otherwise. Then the sequence $\{t_n\}$ is sufficiently different than the sequence (in k) $\{d_{n,k}\}$ in the nth digit, so if t is the number corresponding to $\{t_n\}$, we have that $t \neq s_n$ for all n.

3.2.2 Remarks on the ACL2(r) Formalization

The formalization of this argument in ACL2(r) is mostly straightforward. The function `digit-seq` is constrained to map positive integers to digits, and `digit-seq-sum` converts a portion of this sequence into a number in $[0,1]$. Then we can define the limit of this sum as follows:

```
(defun-std digit-seq-sum-limit ()
 (standard-part (digit-seq-sum 1 (i-large-integer))))
```

To prove that different (enough) sequences correspond to different numbers, we introduce a second constrained function `digit-seq-2` with its own partial sum and limit functions. Then we can carry out the argument as before and show that these limits must be different.

```
(defthm different-enough-digits-implies-different-numbers-of-limit
 (implies (and (posp i)
               (<= (abs (- (digit-seq i) (digit-seq-2 i))) 7)
               (>= (abs (- (digit-seq i) (digit-seq-2 i))) 3))
          (not (equal (digit-seq-sum-limit)
                      (digit-seq-2-sum-limit))))
 :hints ...)
```

To complete the proof, it is only necessary to convert each s_n in the sequence into a sequence of digits, and this can be done with the function `nth-digit`, which is defined as $\lfloor x \cdot 10^n \rfloor \bmod 10$. As before, we define the summation functions `nth-digit-seq-sum` and `nth-digit-seq-sum-limit`, which take partial sums and their limit, respectively. The important lemma is that the limit of these partial sums is the same as the original number that was taken apart by `nth-digit`. This lemma can be proved by finding an upper bound on the difference between the original number and the partial sum up to an arbitrary index k. Now that we can convert a number to a sequence of digits and vice versa, the rest of the proof goes through easily, yielding a second version of the non-denumerability of the reals:

```
(defthm diag-seq-sum-limit-not-in-sequence
 (and (realp (diag-seq-sum-limit))
      (<= 0 (diag-seq-sum-limit))
      (<= (diag-seq-sum-limit) 1)
      (implies (posp i)
               (not (equal (diag-seq-sum-limit) (seq i)))))
 :hints ...)
```

The statement of `diag-seq-sum-limit-not-in-sequence` is typical of theorems in ACL2(r), as it avoids the use of quantifiers. E.g., instead of saying that *some* $x \in [0, 1]$ is not among the $\{s_n\}$, the theorem explicitly names a specific x that is not among the $\{s_n\}$. The way `diag-seq-sum-limit-not-in-sequence` is stated is very much in the tradition of ACL2, as exposed in [5] and [10]. Of course, it is trivial to restate this result using quantifiers, in which case, the final statement of the theorem is almost identical to that in Sect. 3.1. The only difference is that the theorem in this section is specialized for the interval $[0, 1]$, whereas in Sect. 3.1 an arbitrary open interval was permitted.

We close this section by mentioning some differences in the ACL2(r) formalizations of Cantor's two proofs. When we started this project, we were not certain that the second proof could be carried out in ACL2(r), since the argument about the equivalence of sequences and numbers appeared to be significantly different than the usual arguments that have been formalized in ACL2(r). In contrast, we expected the first proof to be much easier to formalize in ACL2(r), since it was based on the notion of completeness, which is directly embedded in ACL2(r) with the function `standard-part`. However, the reverse turned out to be the case.

The first proof limited the use of the nonstandard features of ACL2(r) to the proof that the real numbers are complete. In contrast, the second proof used these features extensively, as they are needed to reason about the equivalence of numbers and infinite sums, as well as the fact that different sums correspond to different numbers. However, the arguments in Cantor's diagonalization proof translated more directly to ACL2(r), very much in the Boyer-Moore tradition. We believe the main reason is that the first proof relied on universally quantified hypotheses—which required the explicit use of quantifiers in ACL2(r)—as well as partially defined functions. Nevertheless, we are pleased to report that the admittedly limited support for quantifiers and partial functions in ACL2(r) was sufficient to formalize both proofs.

4 Existence of Trasncendental Numbers

We conclude this paper with a formalization of Cantor's proof of the existence of transcendental numbers. This turns out to be a corollary of the non-denumerability of the reals, since Cantor proceeds by showing how the algebraic numbers can be enumerated. Some aspects of the proof could be simplified significantly by using more modern arguments. For instance, the fact that the set of polynomials with integer coefficients is denumerable follows directly from the denumerability of words from a finite (even denumerable) alphabet. However, we avoid these modern notions, since our goal is to follow Cantor's argument closely.

4.1 The Informal Argument

A number x is algebraic if there is some nontrivial polynomial P with rational coefficients such that $P(x) = 0$; otherwise, x is called transcendental. It is sufficient to consider polynomials Q with integer coefficients, because if there exists some nontrivial polynomial P with rational coefficients such that $P(x) = 0$, then there must also exists a nontrivial polynomial Q with integer coefficients such that $Q(x) = 0$—just let $Q(x) = q \cdot P(x)$, where q is the product of the denominators of the coefficients of P.

So we wish to show that there is some real number x such that $P(x) \neq 0$ for all polynomials P with integer coefficients. We do so with a counting argument as follows. First, define the height[3] of the polynomial $P = \sum_{i=0}^{n} a_i x^i$ of degree n to be $h(P) = n - 1 + \sum_{i=0}^{n} |a_i|$. Clearly, x^h is of degree h, so there is at least one polynomial for each positive height h. More important, there are only a finite number of polynomials for each height h. This follows, because any polynomial of degree greater than h will have height greater than h. Moreover, a polynomial with a coefficient greater than h or less than $-h$ will have height greater than h. So at most $(2h + 1)^{h+1}$ polynomials can be of height h.

This means that we can enumerate all the polynomials of height h, and this leads to an enumeration of all polynomials with integer coefficients. Simply enumerate the (finite) polynomials of height 1, then the (finite) polynomials of height 2, and so on.

The next step is to use this plan to enumerate the algebraic numbers instead of the polynomials. Simply enumerate the roots of the (finite) polynomials of height 1, then the roots of the (finite) polynomials of height 2, and so on.

Finally, we observe that no sequence of real numbers can completely cover the interval $(0, 1)$ (or any other non-trivial interval), as shown in Sect. 3. That means there is an $x \in (0, 1)$ such that x is not algebraic. I.e., we have shown that there exists at least one transcendental number (and indeed many more).

4.2 Formalizing Polynomials

The first step in the ACL2(r) proof is a formalization of polynomials. We represent polynomials using lists, so that the polynomial $3x^3 + 2x - 6$ is represented as `(-6 2 0 3)`. This allows us to define the function `eval-polynomial` that evaluates a polynomial at a point. The root of a polynomial is then defined as a number x such that `eval-polynomial` returns 0. We can now define what we mean by an algebraic number; i.e, one that is the root of some polynomial with rational coefficients. The definition uses the support for quantifiers in ACL2(r):

[3] There are different notions of "polynomial height"; the one used here is due to Cantor.

```
(defun-sk algebraic-numberp (x)
 (exists poly
        (and (rational-polynomial-p poly)
             (non-trivial-polynomial-p poly)
             (polynomial-root-p poly x))))
```

We will need the fact that a polynomial of degree n has at most n roots. We prove this by dividing a polynomial P by $x-a$ whenever $P(a)=0$. An important lemma is that the resulting quotient is of degree one less than P, as long as P is of degree at least 1 (what we call a "non-trivial" polynomial). Another important lemma is that if $P(b)=0$ and $a \neq b$, then $Q(b)=0$ where Q is the quotient polynomial, i.e., $Q(x)=P(x)/(x-a)$. Once these facts are known, we can show by induction that if P is of degree n, then it has at most n roots.

We now have the tools to find a list containing all the roots of a given polynomial. Although it would be possible to compute many of the (algebraic) roots, it is sufficient to use ACL2(r)'s choice functions to successively add a new root to an existing list of roots.

```
(defchoose choose-new-root (x) (poly roots)
 (and (polynomial-root-p poly x)
      (not (member x roots))))
```

It is then a simple matter to define the function `find-roots-of-poly` which chooses all the roots of a given polynomial. It is trivial to show that `find-roots-of-poly` is of length at most equal to the degree of the polynomial, and that if x is a root of the polymonial, then it must be in the result of `find-roots-of-poly`.

4.3 Enumerating the Algebraic Numbers

We now turn our attention to the enumeration of the algebraic numbers. The first step is to enumerate all the polynomials of height h, and the function `generate-polys-with-height` is defined to do so. The definition is typical of combinatorial functions. I.e., a polynomial p is of degree at most n and height h if either

- $p = ax$ for some constant a and $h = |a|$, or
- p is of degree at most $n-1$ and height h, or
- $p = ax^n + p'$ where $a \neq 0$ and p' is of degree at most $n-1$ and height $h-|a|$.

Since the degree n and leading coefficient a have bounds as explained above, this definition can be implemented recursively. However, the function `generate-polys-with-height` is difficult to introduce into ACL2(r), as many cases need to be considered and it is not obvious why the function terminates—which must be proven before the definition can be accepted. The exact form of the definition is not important, so we omit it here[4]. Instead, we mention the key theorem, namely that the function is guaranteed to generate all the polynomials of the given height.

[4] Interested readers can refer to the supporting materials.

```
(defthm generate-polys-with-height-valid
 (implies (and (integer-polynomial-p poly)
               (non-trivial-polynomial-p poly))
          (member poly (generate-polys-with-height
                        (polynomial-height poly))))
 :hints ...)
```

We can enumerate all the algebraic numbers: We have an enumeration of the polynomials with integer coefficients, so we simply need to find the roots of each polynomial, using `find-roots-of-poly`.

```
(defun enumerate-roots-of-polys (polys)
 (if (consp polys)
     (append (pad-list (length (car polys))
                       (find-roots-of-poly (car polys)))
             (enumerate-roots-of-polys (cdr polys)))
   nil))
```

The function `pad-list` is there for a technical reason. It simply adds zeros to the list of roots, in order to ensure that the list of roots for a polynomial of degree n has $n + 1$ elements, even though the polynomial has fewer (or no) real roots. What this means is that the enumeration returns more than just the list of roots, but this is unimportant. What matters is that all the roots of the polynomials are accounted for. The padding simply makes it easier to associate the ith element of the list of roots with the jth polynomial.

The two functions `enumerate-roots-of-polys` and `generate-polys-with-height` can be used to define `enumerate-roots-of-polys-of-height`, which enumerates the roots of polynomials of the given height. In turn, this can be generalized into `enumerate-roots-of-polys-up-to-height`, which returns all the roots of polynomials of height 1, then those of height 2, and so on, up to a chosen limit:

```
(defun enumerate-roots-of-polys-up-to-height (height)
 (if (zp height)
     nil
   (append (enumerate-roots-of-polys-up-to-height (1- height))
           (enumerate-roots-of-polys-of-height height))))
```

The definition of `enumerate-roots-of-polys-up-to-height` was carefully chosen so that the algebraic numbers come in a predictable order. I.e., calling this function with a higher limit returns additional roots at the *end of* the list, not in the front. We say that the enumeration of `enumerate-roots-of-polys-up-to-height` is monotonic.

Intuitively, if we call `enumerate-roots-of-polys-up-to-height` repeatedly, we will generate all the roots of all polynomials with integer coefficients; i.e., we can enumerate the algebraic numbers. But we have to make this explicit. I.e., we have to define a mapping from the positive integers into the algebraic numbers, and we have to be able to produce the index n such that the mapping yields a particular algebraic number. The mapping can be defined as follows:

```
(defun algebraic-number-sequence (idx)
 (if (posp idx)
     (nth (1- idx) (enumerate-roots-of-polys-up-to-height idx))
   0))
```

Note that the definition uses `enumerate-roots-of-polys-up-to-height` to enumerate all the polynomials up to height `idx`. This works, because of the properties of `enumerate-roots-of-polys-up-to-height` mentioned above. First, we know that the roots of polynomials of height h is non-empty, since there is at least one polynomial of height h (namely x^h) and we are padding the list of roots in the definition of `enumerate-roots-of-polys`. This means that there are at least `idx` roots of polynomials with height at most `idx`, so the call to `nth` in the definition returns a valid element. Second, since the enumeration of `enumerate-roots-of-polys-of-height` is monotonic, it does not matter that the call to `enumerate-roots-of-polys-of-height` uses a height limit (`idx`) that is almost certainly larger than necessary, since there are bound to be many more than one root at each height!

What remains is to show that if `root` is a root of some polynomial with integer coefficients, say `poly`, then there is an index n such that `root` is the nth element in the sequence `algebraic-number-sequence`. We already know that `root` is in `enumerate-roots-of-polys-of-height` h, where h is the height of `poly`, and we can find the index k of `root` in this list using a simple recursive function. Now, let M be the length of `enumerate-roots-of-polys-up-to-height` $h-1$. Then $M+k$ is the index of `root` in `enumerate-roots-of-polys-up-to-height` h' for any $h' \geq h$. To complete the argument, it is only necessary to observe that $M + k \geq h$, and again this follows because there is at least one root at each height. What this means is that $M + k$ is a suitable choice of n, as the following theorem demonstrates, where `get-index-in-last-list` returns $M + k - 1$:

```
(defthm algebraic-number-sequence-valid
 (implies (and (integer-polynomial-p poly)
               (non-trivial-polynomial-p poly)
               (polynomial-root-p poly root))
          (equal (algebraic-number-sequence
                   (1+ (get-index-in-last-list root poly)))
                 root))
 :hints ...)
```

Now we can prove the existence of transcendental numbers. Using the previous theorem, we can show that the sequence `algebraic-number-sequence` contains all the algebraic numbers. Moreover, `algebraic-number-sequence` satisfies the constraints of the function `seq` defined in Sect. 3.1. That means we can apply the main result of that section to conclude that every open interval (for concreteness, the interval $(0, 1)$) must contain at least one number that is not in the sequence. By definition, this number must be transcendental. The final statement of the theorem is as follows:

```
(defun-sk exists-transcendental-number ()
 (exists x
         (and (realp x)
              (not (algebraic-numberp x)))))

(defthm existence-of-transcendental-numbers
 (exists-transcendental-number)
 :hints ...)
```

5 Conclusions

This paper describes a formalization of Cantor's proofs of the non-denumerability of the continuum, a result listed in Freek Wiedijk's Formalizing 100 Theorems [9]. Following Cantor's 1874 paper, this paper also formalizes his proof of the existence of transcendental numbers. The formalization depends on features of ACL2 that are rarely used in ACL2(r), such as choice functions, explicit quantifiers, and partial (or rather underspecified) functions.

Given the two proofs, it would appear that the formalization based on Cantor's familiar diagonalization argument would be the more difficult to formalize in ACL2(r). However, our experience demonstrates that the diagonalization argument can be formalized more directly in the ACL2 (or Boyer-Moore) tradition. It turns out that the original proof is more difficult to formalize in ACL2(r), as it requires explicit use of quantifiers and choice functions.

References

1. Cantor, G.: On a property of the set of real algebraic numbers. In: From Kant to Hilbert, vol. 2, pp. 839–843. Oxford University Press (1874)
2. Ewald, W. (ed.): From Kant to Hilbert, vol. 2. Oxford University Press (2005)
3. Dunham, W.: The Calculus Gallery. Princeton (2005)
4. Cantor, G.: On an elementary question in the theory of manifolds. In: From Kant to Hilbert, vol. 2, pp. 920–922. Oxford University Press (1891)
5. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on Common Lisp. IEEE Transactions on Software Engineering 23(4), 203–213 (1997)
6. Manolios, P., Moore, J.S.: Partial functions in ACL2. Journal of Automated Reasoning (JAR) 31, 107–127 (2003)
7. Nelson, E.: Internal set theory: A new approach to nonstandard analysis. Bulletin of the American Mathematical Society 83, 1165–1198 (1977)
8. Gamboa, R., Kaufmann, M.: Nonstandard analysis in ACL2. Journal of Automated Reasoning 27(4), 323–351 (2001)
9. Wiedijk, F.: Formalizing 100 theorems (2012), `http://www.cs.ru.nl/~freek/100/index.html`
10. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, Orlando (1979)

Construction of Real Algebraic Numbers in Coq

Cyril Cohen

INRIA Saclay–Île-de-France,
LIX École Polytechnique
Microsoft Research - INRIA Joint Centre
cohen@crans.org

Abstract. This paper shows a construction in Coq of the set of real algebraic numbers, together with a formal proof that this set has a structure of discrete Archimedean real closed field. This construction hence implements an interface of real closed field. Instances of such an interface immediately enjoy quantifier elimination thanks to a previous work. This work also intends to be a basis for the construction of complex algebraic numbers and to be a reference implementation for the certification of numerous algorithms relying on algebraic numbers in computer algebra.

Introduction

Real algebraic numbers form the countable subset of real numbers which are roots of polynomials with rational coefficients. This strict sub-field of real numbers has interesting properties that make it an important object for algorithms in computer algebra and in constructive and effective mathematics. For example, they can be substituted for real numbers in the ongoing constructive formalization of Feit-Thompson Theorem. Indeed, there is an effective algorithm to compare two algebraic numbers and all field operations can be defined in an exact way. Moreover, they can be equipped with a structure of discrete Archimedean real closed field, which is an Archimedean ordered field with decidable ordering satisfying the intermediate value property for polynomials.

The aim of this paper is to show how we define in Coq a data-type representing the real algebraic numbers and to describe how to formally show it is an Archimedean real closed field. This construction and these proofs are described in many standard references on constructive mathematics [11] or in computer algebra [2]. However, the implementation of these results in a proof assistant requires various changes in their presentation. Hence our development is not a literate translation of a well-chosen reference, but is rather a synthesis of results from the mathematical folklore which are often unused in the literature because they are subsumed by classical results.

In order to define real algebraic numbers, standard references usually suggest one of the following strategies. The first one takes a type representing real numbers and builds the type representing the subset of reals which are roots of a polynomial with rational coefficients. One must then show that induced arithmetic operations on this subset have the expected properties. The second strategy starts from a type representing rational numbers and formalizes the real closure of rational numbers, which is the smallest real closed field containing them. An element of the closure is usually represented as a pair polynomial - interval, satisfying the invariant that the polynomial has a unique root in the interval. This selected root is the algebraic number encoded by that pair. From a constructive point of view, there is no reason to prefer one or the other of these strategies: it may of course be possible to complete the required proofs in any of these two cases. However, there are significant differences in the nature of objects and proofs we handle when formalized in type theory.

In this work, we combine the two approaches in order to get the advantages of both and to eliminate their respective drawbacks.

Constructive formal libraries on exact reals are available in the Coq system [8]. However, for the requirements of this formalization we developed a short library constructing exact reals as Cauchy sequences from an arbitrary Archimedean field. We explain these formalization choices and our construction in Section 2.

Then, in Section 3 we introduce a first type for algebraic real numbers which we call algebraic Cauchy reals, together with its comparison algorithm and arithmetic operations. In particular, we show how to compute annihilating polynomials, decide the equality and more generally the comparison.

We then describe in Section 4 how to construct the real closure of rational numbers to get a second data-type for real algebraic numbers, that we call real algebraic domain.

Thanks to this second data-type and to the equality decision procedure, we show in Section 5 how to form the real algebraic numbers and we prove that it is a real closed field. The key ingredient is the proof of the intermediate value property for polynomials, which concludes this work.

The complete Coq formalization we describe in this paper is available at `http://perso.crans.org/cohen/work/realalg`. The code excerpts of the paper may diverge from the actual code, for the sake of readability. However, we wrote the proofs in a way which is very close to their Coq formalization.

1 Preliminaries

In this work, we use the SSReflect library of the *Mathematical Components* project [13]. We base our development on the algebraic hierarchy [7], with the extensions we already brought to describe discrete ordered structures [5]. We use mostly the discrete real closed field structure. We also take advantage of the available libraries on polynomials with coefficients in rings or fields. More precisely, we use the polynomial arithmetic library which grants the following definitions and properties: arithmetic operations, euclidean division, Bézout theorem, Gauss theorem.

We explain in more details some elements of the SSReflect library we use.

In the SSReflect library, algebraic structures are equipped with a decidable equality and a choice operator.

Decidable Equality Structure

Decidable equality structures are instances of an interface called `eqType`. Such a structure is a dependently typed record that bundles a type, together with a boolean relation `(eq_op : T →T →bool)` and a proof it reflects the Leibniz equality, which means:

```
∀(T : eqType) (x y : T), x = y ↔(eq_op x y = true)
```

The SSReflect library provides a rich theory about `eqType`, such as for example the uniqueness of equality proofs on such types. The importance of this structure also comes from the SSReflect methodology to go back and forth between boolean statements and propositional statements in order to alternate computational steps with deductive steps.

Choice Structure

Choice structures are instances of an interface called `choiceType` in the library. They provide us the choice operator `xchoose` of type:

```
xchoose : ∀(T : choiceType) (P : T →bool), (∃x, P x) →T.
```

which satisfies the two following properties :

```
xchooseP : ∀(T : choiceType) (P : T →bool) (xP : ∃x, P x),
   P (xchoose T P xP).
eq_xchoose : ∀(T : choiceType) (P Q : T →bool)
                (xP : ∃x, P x) (xQ : ∃x, Q x),
   (∀x, P x = Q x) →xchoose T P xP = xchoose T Q xQ.
```

which respectively ensure the correctness and uniqueness of the chosen element with respect to the predicate `P`.

For instance, in Coq, any countable type can be provably equipped with such a structure. This means we can take T to be the type $\mathbb{Q}$ of rational numbers.

The choice structure is fundamental to formalize both the comparison of Cauchy reals in Section 2.2 and the construction of the effective quotient type in Section 5.

Resultant of Two Polynomials and Corollary to Bézout Theorem

The resultant of two polynomials $P = \sum_{i=0}^{m} p_i X^i$ et $Q = \sum_{i=0}^{n} q_i X^i$ is usually defined as the determinant of the Sylvester matrix.

$$\mathrm{Res}_X(P,Q) = \begin{vmatrix}
p_m & p_{m-1} & \cdots & p_0 & 0 & 0 & \cdots & 0 \\
0 & p_m & p_{m-1} & \cdots & p_0 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & p_m & p_{m-1} & \cdots & p_0 & 0 \\
0 & \cdots & 0 & 0 & p_m & p_{m-1} & \cdots & p_0 \\
q_n & q_{m-1} & \cdots & q_0 & 0 & 0 & \cdots & 0 \\
0 & q_n & q_{m-1} & \cdots & q_0 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & q_n & q_{m-1} & \cdots & q_0 & 0 \\
0 & \cdots & 0 & 0 & q_n & q_{m-1} & \cdots & q_0
\end{vmatrix}$$

The notion of resultant is well described and studied in numerous books, we invite the reader to look in one of them, for instance in [10]. If the polynomials are univariate the resultant is a scalar, if they are bivariate, it is a univariate polynomial in the remaining variable. In our development we use only the two following properties of the resultant $\mathrm{Res}_X(P(X,Y),Q(X,Y)) \in F[Y]$ of polynomials $P,Q \in F[X,Y]$ where F is a field:

$$\exists U,V \in F[X,Y], \quad \mathrm{Res}_X(P,Q) = UP + VQ$$
$$\mathrm{Res}_X(P,Q) = 0 \quad \Leftrightarrow \quad P \text{ and } Q \text{ are not coprime as polynomials in } X$$

which respectively express that the resultant of P and Q is in the ideal generated by P and Q, and is zero if and only if P and Q are not coprime as polynomials in X with coefficients in $F[Y]$, i.e. they have no common factor in $(F[Y])[X]$.

Moreover we use the following corollary to Bézout theorem: If P and Q are not coprime as polynomials in X with coefficients in $F[Y]$, there exist U and V in $F[X,Y]$ such that U is non zero, $\deg_X(U) < \deg(Q)$ and

$$U(X,Y)P(X,Y) = V(X,Y)Q(X,Y)$$

2 Construction and Properties of Cauchy Reals

From now on, we denote by F an ordered Archimedean field equipped with a decidable equality structure and with a choice structure. All the constructions are done over F which is, for our purpose, an appropriate generalization of $\mathbb{Q}$. Although it is necessary for this construction, we do not detail the use of the Archimedean property for the sake of readability.

It remains unclear whether an axiomatization of Cauchy reals as described in [8] would fit our needs. Moreover, our implementation is shorter and more direct, but less generic, when compared with Russell O'Connor's [12].

2.1 Mathematical Description and Coq Data-Type

We define a Cauchy real as a sequence $(x_n)_{n\in\mathbb{N}}$ in $F^{\mathbb{N}}$, together with a convergence modulus $m_x : F \to \mathbb{N}$ such that from the index $m_x(\varepsilon)$, the distance between any two elements is smaller than ε. This "Cauchy property" is stated as:

$$\forall\varepsilon\ \forall i\ \forall j\ ,\ m_x(\varepsilon) \le i\ \wedge\ m_x(\varepsilon) \le j \Rightarrow |x_i - x_j| < \varepsilon$$

We encode sequences of elements of F as functions from natural numbers to F. Hence, we encode Cauchy reals by packaging together the sequence $(x_n)_n$, the modulus m_x and the "Cauchy property":

```
Definition creal_axiom (x : nat →F) :=
  {m : F →nat | ∀ε i j, m ε ≤ i →m ε ≤ j →|x i - x j| < ε}.
Inductive creal := CReal
  {cauchyseq : (nat →F);      _ : creal_axiom cauchyseq}.
```

We remind that `{m : F →nat | ...}` is called a sigma-type and can be read "there exist a function `(m : F →nat)` such that".

The C-CoRN library also provides an interface for Cauchy reals and a construction of Cauchy sequences, which is used to instantiate the interface in [8]. Although their definition is close enough to ours, we redefine and re-implement Cauchy reals from scratch, mainly because our algebraic structures are incompatible. We use this as an opportunity to restate the definitions in a way which is more compatible with our proof style.

In this paper, we often denote a Cauchy sequence $(x_n)_n$ of convergence modulus m_x by the notation $\bar{x}$. We call such an element a Cauchy real and it represents a constructive real number. We often take the ith element of the underlying Cauchy sequence of $\bar{x}$, and we denote it as x_i. Moreover, in Coq code, m_x is encoded as a function `(cauchymod x)` of type `(F →nat)`. A Coq user will remark that such a function is definable because the existential modulus in the definition of the Cauchy sequence is in `Type`.

By definition of Cauchy sequences, we get the following property:

```
Lemma cauchymodP (x : creal) (ε : F) (i j : nat) :
  cauchymod x ε ≤ i →cauchymod x ε ≤ j →|x i - x j| < ε
```

It is important to note that when we apply this lemma, we produce a sub-goal (which we call side condition) of the form $f(\varepsilon) \leq i$. This is a general scheme in our development: during a proof we may generate n side conditions $f_k(\varepsilon) \leq i$ for $k \in \{1, \ldots, n\}$. Indeed, if all constraints on i are formulated like this, it suffices to take i to be the maximum of all the $f_k(\varepsilon)$, in order to satisfy all the side conditions on i. We even have designed an automated procedure to solve this kind of constraints using the `Ltac` language [6] available in Coq, so that many proofs begin with a command meaning "let i be a big enough natural number".

From `cauchymod` we can define a function ubound to bound above the values of elements of a Cauchy sequence. It then satisfies the following property:

```
Lemma uboundP : ∀(x : creal) (i : nat), |x i| ≤ ubound x.
```

In the rest of the development, this function is used to compute the convergence moduli of numerous Cauchy sequences. We use the notation $\lceil x \rceil$ for `(ubound x)`.

2.2 Comparison

On Cauchy reals, the Leibniz equality is not a good notion to compare numbers, as two distinct sequences may represent the same real number. In fact, the good correct of equality on Cauchy reals states that $\bar{x}$ and $\bar{y}$ are equivalent if the sequence of point-wise distances $(|x_n - y_n|)_n$ converges to 0.

A type together with an equivalence relation is called a setoid, and the equivalence is the setoid equality. Coq provides tools to declare setoids, to declare functions that are compatible with the setoid equality, and eventually to rewrite using the setoid equality in contexts that are compatible with it [1].

Although the comparison of Cauchy reals is not decidable, telling whether $\bar{x}$ and $\bar{y}$ are distinct is semi-decidable: classically, if they are not equal, there exist a quantity δ and an index k such that $\delta \leq |x_i - y_i|$ for all i greater than k. Hence the primitive notion for comparison is not equality but apartness, which contains additional information: a witness for the non-negative lower bound of the gap separating the two sequences.

For the sake of clarity we write $\bar{x} \neq \bar{y}$ for apartness and $\bar{x} \equiv \bar{y}$ for its negation. The notion of non apartness coincides with the notion of equivalence stated above and is declared as the setoid equality on Cauchy reals.

From a proof of apartness $\bar{x} \neq \bar{y}$ we must be able to extract a rank k and a non-negative witness δ which bounds below the sequence $(|x_n - y_n|)_n$ from the rank k. This lower bound is needed to define the inverse as described in Section 2.4. So we could define apartness as follows, using a witness in `Type` to make it available for computation:

```
Definition bad_neq_creal x y : Type := {δ : F | 0 < δ &
∀i, cauchymod x δ ≤ i →cauchymod y δ ≤ i →δ ≤ |x i - y i|}.
```

But to be fully compatible with the setoid mechanism, the apartness must be in `Prop`, not in `Type`. Robbert Krebbers and Bas Spitters [9] already encountered this problem in C-CoRN and solved it using the "constructive indefinite description" theorem, which is provable for decidable properties whose domain is `nat`. Our solution uses a variant of this theorem, thanks to the `choiceType` structure of `F`.

We define apartness ($\neq$) as follows:

```
Definition neq_creal (x y : creal) : Prop :=
  ∃δ, (0<δ) && (3 * δ ≤ |x (cauchymod x δ) - y (cauchymod y δ)|).
```

Then, using `xchoose`, we can define the non-negative lower bound function:

```
Definition lbound x y (neq_xy : x ≠y) : F := xchoose F _ neq_xy.
```

Given two Cauchy reals $\bar{x}$ and $\bar{y}$ which are provably apart from each other, let δ be their non-negative lower bound of separation as defined above. From `xchooseP` we get that $3\delta \leq \left|x_{m_x(\delta)} - y_{m_y(\delta)}\right|$. Thus:

$$\forall i, \quad 3\delta \leq \left|x_{m_x(\delta)} - x_i\right| + |x_i - y_i| + \left|y_i - y_{m_y(\delta)}\right|$$

But since we work on Cauchy sequences, we know how to bound the distance between any two elements of the sequence, starting from a well chosen index: $\forall\, i \geq m_x(\delta),\ \left|x_{m_x(\delta)} - x_i\right| < \delta$ and $\forall\, i \geq m_y(\delta),\ \left|y_i - y_{m_y(\delta)}\right| < \delta$. So:

$$\forall\, i \geq \max(m_x(\delta), m_y(\delta)), \quad \delta \leq |x_i - y_i|$$

Hence we prove the lemma:

```
Lemma lboundP (x y : creal) (neq_xy : x ≠ y) i :
  cauchymod x (lbound neq_xy) ≤ i →
  cauchymod y (lbound neq_xy) ≤ i → lbound neq_xy ≤ |x i - y i|.
```

2.3 Order Relation

The order relation is handled the same way as apartness. The primitive notion is the strict ordering, the negation of which defines the non-strict ordering. For the sake of space we don't write much about comparison as beyond noting it is derivable from a proof of apartness:

```
Lemma neq_ltVgt (x y : creal) : x ≠ y → {x < y} + {y < x}.
```

where the operator `+` is the disjunction in `Type`.

2.4 Arithmetic Operations on Cauchy Reals

We build the negation, addition and multiplication on Cauchy reals and prove their output are Cauchy sequences in a systematic way: we perform the appropriate operation on each element of the sequence and we forge a convergence modulus for each operation.

To build the negation, addition and multiplication, we exhibit the convergence moduli of negation, addition and multiplication of Cauchy reals. Given the convergence modulus m_x of $\bar{x}$, we prove the convergence moduli of $(-x_n)_n$, $(x_n + y_n)_n$ and $(x_n y_n)_n$ are respectively: m_x, $\varepsilon \mapsto \max\left(m_x\left(\frac{\varepsilon}{2}\right), m_y\left(\frac{\varepsilon}{2}\right)\right)$ and $\varepsilon \mapsto \max\left(m_x\left(\frac{\varepsilon}{2\lceil y\rceil}\right), m_y\left(\frac{\varepsilon}{2\lceil y\rceil}\right)\right)$

To build the inverse, we need to know a non-negative lower bound δ for the sequence $(|x_n|)_n$ of absolute values from some arbitrary rank, and use it to prove that the sequence of point-wise inverses $(\frac{1}{x_n})_n$ is a Cauchy sequence. According to Section 2.2, such a non-negative lower bound δ is given by `(lbound x_neq0)` when given a proof `(x_neq0 : x ≠ 0)` that $\bar{x}$ is apart from 0 (in the sense of Cauchy sequences). This value δ is such that $\forall i > m_x(\delta)$, $\delta \leq |xi|$

If i and j are greater than $m_x(\varepsilon\delta^2)$, we have $|x_i - x_j| < \varepsilon\delta^2$ By definition of δ and if i and j are greater than $m_x(\delta)$, we get $\delta \leq |x_i|$ and $\delta \leq |x_j|$, thus $|x_i - x_j| < \varepsilon|x_i x_j|$. And finally:

$$\left|\frac{1}{x_i} - \frac{1}{x_j}\right| < \varepsilon$$

Thus, a convergence modulus is $\varepsilon \mapsto \max\left(m_x(\varepsilon\delta^2), m_x(\delta)\right)$

Morphism property of arithmetic operations. We can check that all arithmetic operations are compatible with the equality for Cauchy sequences, using a simple point-wise study. The order relation is also a compatible. However, there is no need to systematically study the compatibility with apartness.

2.5 Bounds and Evaluation for Polynomials

Using the Taylor expansion of polynomial P, we define the following bounds:

$$B_0(P,c,r) = 1 + \sum_{i=0}^{n} |p_i|(|c|+|r|)^i$$

$$B_1(P,c,r) = \max(1,2r)^n \left(1 + \sum_{i=1}^{n} \frac{B_0(P^{(i)},c,r)}{i!}\right)$$

$$B_2(P,c,r) = \max(1,2r)^{n-1} \left(1 + \sum_{i=2}^{n} \frac{B_0(P^{(i)},c,r)}{i!}\right)$$

These bounds satisfy the following properties, for all x and y in $[c-r,c+r]$:

$$|P(x)| \leq B_0(P,c,r)$$

$$|P(y)-P(x)| \leq |y-x|B_1(P,c,r)$$

$$\left|\frac{P(y)-P(x)}{y-x} - P'(x)\right| \leq |y-x|B_2(P,c,r)$$

These bounds are constructive witnesses for well-known classical mathematical results on continuous or derivable functions, specialized to univariate polynomials. The bound B_0 is only an intermediate step to bounds B_1 and B_2. The bound B_2 is used in Section 4.2 to prove that polynomials whose derivative does not change sign on an interval are monotone on it.

The bound B_1 is used to show that polynomial evaluation preserves the Cauchy property for sequences. Indeed, we build polynomial evaluation of a polynomial $P \in F[X]$ in a Cauchy real as the point-wise operation, and in order to prove that the result is a Cauchy sequence, we bound $|P(x)-P(y)|$ when $|x-y|$ is small enough. The convergence modulus is given by $\varepsilon \mapsto m_x\left(\frac{\varepsilon}{B_1(P,0,\lceil x \rceil)}\right)$. We then prove that $P(\bar{x}) \neq P(\bar{y}) \Rightarrow \bar{x} \neq \bar{y}$, which implies that $\bar{x} \equiv \bar{y} \Rightarrow P(\bar{x}) \equiv P(\bar{y})$, hence the evaluation of a polynomial in a Cauchy real is compatible with the equality of Cauchy reals.

3 An Existential Type for Algebraic Cauchy Reals

3.1 Construction of Algebraic Cauchy Reals

Now, we formalize real algebraic numbers on top of Cauchy reals.

```
Inductive algcreal := AlgCReal {
  creal_of_alg : creal;
  annul_algcreal : {poly F};
  _ : monic annul_algcreal;
  _ : annul_algcreal.[creal_of_alg] ≡0
}.
```

Here, an algebraic Cauchy real `(AlgCReal x P monic_P root_P_x)` represents an algebraic number as a Cauchy real `x` and a polynomial `P` with a proof `monic_P` that `P` is monic (its leading coefficient is 1) and a proof `root_P_x` that `x` is a root of `P`. The notation `p.[x]` stands for polynomial evaluation in the source code.

First we prove that Cauchy reals setoid equality is decidable on algebraic Cauchy reals, then we build arithmetic operations.

3.2 Equality Decision Procedure

Whereas the comparison on Cauchy reals is only semi-decidable, the comparison on algebraic Cauchy reals is decidable. We call `eq_algcreal` this decision procedure. It uses the additional data given by the annihilating polynomials. In fact, we only need to decide if some algebraic Cauchy real is zero, because we can test whether $\bar{x} = \bar{y}$ by comparing $\bar{x} - \bar{y}$ to 0 once we have the subtraction.

Let $(\bar{x}, P)$ be an algebraic Cauchy real we wish to compare to 0, so P is the annihilating polynomial of the Cauchy real $\bar{x}$. There are two possibilities:

– *Either the indeterminate X does not divide P*, then 0 is not a root of P, thus $\bar{x} \neq 0$.
– *Or X divides P*. If $P = X$ then $\bar{x} \equiv 0$, so let us suppose that X is a proper divisor of P. Then there exist a divisor D of P whose degree is smaller than the one of P and such that $D(\bar{x}) \equiv 0$. The existence of such a D is given by a general lemma stating that if $\bar{x}$ is a Cauchy real and P, Q two polynomials that are not coprime and such that $P(\bar{x}) \equiv 0$ and P does not divide Q, then there exist D of smaller degree than P such that $D(\bar{x}) \equiv 0$.
  We can now iterate this reasoning on $(\bar{x}, D)$ where the degree of D is smaller than the one of P.

3.3 Operations on Algebraic Cauchy Reals

We build all the operations (negation, addition, multiplication, inverse) from the constants 0 and 1 and using the subtraction and the division. The embedding of the constants $c \in F$ is obtained from the pair $(\bar{c}, X - c)$ (where $\bar{c}$ is a constant Cauchy sequence).

In the remainder of this section we consider two algebraic Cauchy reals x and y, whose respective Cauchy sequences are $\bar{x}$ and $\bar{y}$, and whose respective annihilating polynomials are P and Q.

Let us recall (Section 2.4) that the subtraction $\bar{x} - \bar{y}$ (resp. division $\frac{\bar{x}}{\bar{y}}$) is obtained as the point-wise subtraction (resp. division) of elements of the sequence. Let us find a polynomial whose root is this new sequence.

Subtraction. Our candidate is the following resultant:

$$R(Y) = \mathrm{Res}_X\,(P(X+Y), Q(X))$$

There are two essential properties to prove about this resultant it is non zero and it annihilates the subtraction.

R is non zero. Let us suppose that R is zero and find a contradiction. Since R is zero, $P(X+Y)$ and $Q(X)$ are not coprime.

Thanks to the corollary to Bézout theorem, we know there exist $U, V \in F[X]$ such that U is non zero, $\deg_X(U) < \deg(Q)$ and $U(X,Y)P(X+Y) = V(X,Y)Q(X)$.

Taking the Y-leading coefficient, we get $u(X)p = v(X)Q(X)$ where $u(X)$ and $v(X)$ are the respective Y-leading coefficients of $U(X,Y)$ and $V(X,Y)$, and p is the leading coefficient of P. This equation gives that $\deg(Q) \leq \deg(u)$, but $\deg(u) \leq \deg_X(U) < \deg(Q)$. This is a contradiction.

R annihilates the subtraction. Let us prove that R annihilates the Cauchy sequence $\bar{x} - \bar{y}$. Since R is in the ideal generated by $P(X+Y)$ and $Q(X)$, there exist U and V such that $R(Y) = U(X,Y)P(X+Y) + V(X,Y)Q(X)$. Hence by evaluation at $X = y_n$ and $Y = (x_n - y_n)$:

$$R(x_n - y_n) = U(y_n, x_n - y_n)P(x_n) + V(y_n, x_n - y_n)Q(y_n)$$

But $P(\bar{x}) \equiv 0$ and $Q(\bar{y}) \equiv 0$. As x_n and y_n are bounded and U is bounded on a bounded domain (cf Section 2.5) we have that $R(\bar{x} - \bar{y}) \equiv 0$.

Remark that now the subtraction is defined, we can decide the equality of two arbitrary values by comparing their subtraction to zero, using the result from Section 3.2.

Division. When $\bar{y}$ is zero, we return the annihilating polynomial X. When it is non zero, we can find a new Q annihilating $\bar{y}$ such that $Q(0) \neq 0$. The annihilating polynomial of $\frac{\bar{x}}{\bar{y}}$ is the following resultant:

$$R(Y) = \mathrm{Res}_X\left(P(XY), Q(X)\right)$$

R is non zero. Let us suppose that R is zero and find a contradiction. Since R is zero, $P(XY)$ and $Q(X)$ are not coprime.

Thanks to the corollary to Bézout theorem, we know there exist $U, V \in F[X]$ such that U is non zero, $\deg_X(U) < \deg(Q)$ and $U(X,Y)P(XY) = V(X,Y)Q(X)$.

By evaluation at $Y = 0$ we get: $U(X,0)P(0) = V(X,0)Q(X)$. Since $F[Y]$ is an integral domain, if $V(X,0) = 0$ we know that $Y|V(X,Y)$, and that there are two possibilities:

- Either $U(X,0) = 0$, which means $Y|U(X,Y)$. Hence, there exists $U'(X,Y)$ and $V'(X,Y)$, whose degrees in Y are strictly smaller than the ones of U and V, and such that: $U'(X,Y)P(XY) = V'(X,Y)Q(X)$.
- Or $P(0) = 0$, which means $X|P(X)$, thus $XY|P(XY)$. But we also know that $U(0,Y)P(0) = V(0,Y)Q(0)$. And since $Q(0) \neq 0$, we necessarily have $V(0,Y) = 0$. It follows that $X|V(X,Y)$ and as we knew that $Y|V(X,Y)$, we find that $XY|V(X,Y)$.

  Thus, there exist P' and V' whose degrees are strictly smaller than those of P and V respectively, such that $U(X,Y)P'(XY) = V'(X,Y)Q(X)$.

In both cases, we can repeat the same reasoning until we get an equation of the following form, such that no member cancels: $U(X,0)P(0) = V(X,0)Q(X)$. This equation gives $\deg(Q) \leq \deg(U(X,0))$, but we also had $\deg(U(X,0)) \leq \deg_X(U) < \deg(Q)$. This is a contradiction.

R annihilates the division. In the same way we did for subtraction, we show that $R(\frac{\bar{x}}{\bar{y}}) \equiv 0$.

4 Encoding Algebraic Cauchy Reals

The data-type of algebraic Cauchy reals is a setoid whose equivalence is decidable, and it is difficult to show that algebraic Cauchy reals form a countable setoid if F is countable. However, we can do better and build a type whose decidable equivalence reflects Leibniz equality, and for which we can exhibit a bijection with $\mathbb{N}$ if F is countable.

In order to get the type of real algebraic numbers, we should quotient the type of algebraic Cauchy reals by the setoid equality. We know from [3] that this quotient can be done inside Coq as soon as the type which gets quotiented has a `choiceType` structure and the equivalence relation by which we quotient is decidable. Since `algcreal` cannot directly be equipped with a `choiceType` structure, we create a type `algdom` which we call real algebraic domain. The type `algdom` only serves as an encoding of `algcreal` in order to forge the quotient, the construction of which we detail in Section 5.

```
Inductive algdom := AlgRealDom {
  annul_algdom : {poly F};
  center_alg : F;
  radius_alg : F;
  _ : monic annul_algdom;
  _ : annul_algdom.[center_alg - radius_alg]
      * annul_algdom.[center_alg + radius_alg] ≤ 0
}.
```

An element `(AlgRealDom P c r monic_P chg_sign_P)` of `algdom` represents one of the roots of the polynomial `P` in the interval `[c - r, c + r]`, with a proof `monic_P` that `P` is monic and a proof `chg_sign_P` that `P` changes sign on the interval. We know which root is selected by running the decoding procedure described in Section 4.1.

This data-type is only using elements of F and two proofs. It thus can be encoded as sequences of elements of F and inherits the `choiceType` structure of F. We also notice that `algdom` is countable as soon as F is. This fact was not obvious for the setoid of algebraic Cauchy reals. The quotient type will also inherit from the `choiceType` structure and will be countable if F is.

We show that `algdom` is an explicit encoding of algebraic Cauchy reals. Remark that `algcreal` is still useful because arithmetic operations are easier to define on it.

4.1 Decoding to Algebraic Cauchy Reals

We build the decoding function `to_algcreal`: `algdom` $\rightarrow$ `algcreal`.

An element from the real algebraic domain contains a polynomial P, a center c and a radius r such that $P(c-r)P(c+r) \leq 0$. The root we wish to select is in the interval $I = [c-r, c+r]$.

We decode an element from the real algebraic domain into an algebraic Cauchy real by dichotomy. We form the Cauchy sequence $\bar{x} = (x_n)_n$, such that all the x_n are in the interval I and such that $P(\bar{x}) \equiv \bar{0}$.

We proceed by induction on n to define the sequence $\bar{x}$. It should satisfy the following invariant, which expresses that P must change sign on the interval of radius $2^{-n}r$ and centered in x_n:

$$H_n = P(x_n - 2^{-n}r)P(x_n + 2^{-n}r) \leq 0$$

In the induction step, we pick either $x_n - 2^{-(n+1)}r$ or $x_n + 2^{-(n+1)}r$ to satisfy the invariant H_{n+1}.

The condition that it changes sign is sufficient to show the existence of a root, and doesn't assert anything about its unicity. However, we have no need for unicity as the decoding procedure selects a root in a deterministic manner.

4.2 Encoding of Algebraic Cauchy Reals

This step is more difficult, we construct the encoding function `to_algdom`: `algcreal` $\rightarrow$ `algdom`. In order to satisfy the coding property:

```
Lemma to_algdomK x : to_algcreal (to_algdom x) ≡ x.
```

Given an algebraic Cauchy real $(\bar{x}, P)$ we try to find a rational interval containing only one root, in order for the decoding to return an element equivalent to $\bar{x}$.

There are two possibilities:

- *Either P and its derivative P' are coprime,* so there exist U and V such that $UP + VP' = 1$. Since $P(\bar{x})$ converges to 0 and if n is big enough we get $P'(x_n) \geq \frac{1}{2\lceil V(\bar{x})\rceil}$. By taking a small enough interval $[a, b]$ containing x_n, we get that P is monotone on $[a, b]$ (thanks to the B_2 bound of Section 2.5) Without loss of generality, we can suppose that P is increasing, we then get $P(a) \leq P(x_i) \leq P(b)$ for all $i \geq n$. But $P(x_i)$ converges to 0, so $P(a) \leq 0 \leq P(b)$. We found an interval with only one root for P.
- *Or P and P' are not coprime,* so we can find a proper divisor D of P that still annihilates x, thanks to the same general lemma mentioned in Section 3.2, in the second case of the disjunction. We fall back to the study of $(\bar{x}, D)$, where the degree of D is strictly smaller that the one of P.

4.3 Transferring the Operations to the Encoding

We can transpose all the operations and properties of algebraic Cauchy reals to its encoding real algebraic domain. More particularly, equality between algebraic

Cauchy reals $\equiv$ (which we showed decidable in 3.2) gives a decidable equivalence on real algebraic domain, using the following definition:

```
eq_algdom x y := (eq_algcreal (to_algcreal x) (to_algcreal y))
```

All the properties of these new operators are easily derived from the properties of the original operators.

5 Real Algebraic Numbers as a Quotient Type

The construction of the quotient is done in a generic way, but for this paper to be self-contained, we describe its construction as it is automatically done by the mechanism presented in [3].

5.1 Construction of the Quotient Type

First we define a notion of canonical element. To each element `x` in `algdom`, we associate an element `(canon x)` which must be equal to any `(canon y)` if and only if `eq_algdom x y`. We use the unique choice operator `xchoose` to do this:

```
Lemma exists_eq (y : algdom) : ∃x : algdom, y ≡x.
Proof. exists y; reflexivity. Qed.

Definition canon (y : algdom) = xchoose (exists_eq y).
```

Moreover, `canon` is constant on each equivalence class thanks to the unicity property of `xchoose`.

Then we define the quotient type of real algebraic numbers by forming the sigma-type of elements of the real algebraic domain that are canonical:

```
Definition alg := {x : algdom | canon x = x}
```

Thanks to the uniqueness of equality proofs on `algdom`, two elements `x` and `y` in `alg` are equal if and only if `(val x = val y)`, where `val` is the projection on the first component of the sigma-type. From `canon`, we can now build the canonical surjection (`pi : algdom →alg`), which maps any element of `algdom` to the unique representative for its equivalence class.

By composing `to_algdom` with `pi` we can now see `alg` as the type of equivalence classes of elements of `algcreal`. We now see `F` as a parameter for the whole construction, so that `alg` becomes `(alg F)`, which we denote by $\bar{F}$.

We prove that arithmetic operations (and the order relation) are compatible with the quotient. This is a direct consequence of the morphism property of operations with regard to setoid equality, which we dealt with in Section 3.3.

We also build a function (`to_alg: F →alg F`) which embeds any element c of F into $\bar{F}$, by mapping c to the equivalence class of the element $(\bar{c}, (X - c))$ of `algcreal`. We then prove it is a field morphism and that this morphism is also compatible with comparison. The mathematical notation for this function is $\uparrow$.

We remark that by construction of `algdom`, the following property holds: given a polynomial $P \in F[X]$ and two points $a < b \in F$ such that $P(a) \leq 0 \leq P(b)$, there exist $c \in \bar{F}$ such that $c \in [a, b]$ and $P(c) = 0$. This is a weak version of the intermediate value property for polynomials.

5.2 Real Algebraic Numbers Form a Real Closed Field

Note that $\bar{F}$ is a totally ordered Archimedean field with decidable comparison. Indeed, as those properties already hold for F, they transfer to $\bar{F}$ by studying the Cauchy sequences underlying its elements.

The difficulty is to prove $\bar{F}$ is a real closed field, which amounts to prove the intermediate value theorem for polynomials in $\bar{F}[X]$.

Let P be a polynomial in $\bar{F}[X]$ and a and b two elements of $\bar{F}$ such that $a < b$ and $P(a) \leq 0 \leq P(b)$. Let us show that there exist an real algebraic number c in $\bar{F}$ such that $P(c) = 0$.

Iteration of the Closure. Thanks to the remark in the end of Section 5.1, applied to the ordered Archimedean field $\bar{F}$, we get that the polynomial $P \in \bar{F}[X]$ has a root ξ in the "double closure" $\bar{\bar{F}}$.

If we find a function $\downarrow : \bar{\bar{F}} \to \bar{F}$, such that $\forall \zeta \in \bar{\bar{F}}, \quad \uparrow (\downarrow \zeta) = \zeta$, then $(\downarrow \xi) \in \bar{F}$ would be a root of P. The Coq name for this function is `from_alg`. The existence of such a function means that the closure process we design terminates in one step only.

Let ξ be in $\bar{\bar{F}}$, and let us build $(\downarrow \xi)$. By transforming ξ in an algebraic Cauchy real $(\bar{\xi}, P)$ we get a Cauchy sequence $\bar{\xi}$ in $\bar{F}^{\mathbb{N}}$, and a polynomial $P \in \bar{F}[X]$.

Each element ξ_n is a Cauchy sequence $\bar{x}_n = (x_{n,k})_k$ which we can choose such that $|\bar{x}_{n+1} - \bar{x}_n| < 2^{-(n+1)}$. Then, the sequence $\bar{x} = (x_{n,n})_n$ is a Cauchy sequence such that $\uparrow \bar{x} = \bar{\xi}$. We hence have the first component of $(\downarrow \xi)$.

Polynomial Annihilating the Algebraic Cauchy Real $\bar{x}$. We must find a polynomial $R \in F[X]$ which annihilates $\bar{x}$. The coefficients p_i of P are a finite number of values in the field extension $\bar{F}$ of F, so we can apply the primitive element theorem to find an element $\alpha \in \bar{F}$, whose annihilating polynomial is Q of degree $q + 1$ such that for all i, p_i is in the simple extension $F[\alpha]$. We can then re-factorize P as $P = \sum_{l=0}^{q} \alpha^l P_l$.

We take the resultant $R(Y) = \mathrm{Res}_X \left(\sum_{l=0}^{q} X^l P_l(Y), Q(X) \right)$. We now show that it is non zero and it annihilates $\bar{x}$.

R is non zero. Let us suppose R is zero and find a contradiction. The property of Bézout gives $U, V \in F[X]$ such that U is non zero, $\deg_X(U) < \deg(Q)$ and:

$$U(X, Y) \sum_{l=0}^{q} X^l P_l(Y) = V(X, Y) Q(X)$$

Then by embedding in $\bar{F}$ and evaluation at $X = \alpha$ we get: $U(\alpha, Y)P(Y) = 0$. But $P \neq 0$, thus $U(\alpha, Y) = 0$. Then by taking the Y-leading coefficient $u(X)$ of $U(X, Y)$ we get:

$$u(\alpha) = 0 \quad \text{and} \quad u \in F[X] \quad \text{and} \quad u \neq 0 \quad \text{and} \quad \deg(u) < \deg(Q)$$

This gives a polynomial u annihilating α of degree smaller than the one of Q, and we can proceed by induction on the degree of Q.

R annihilates $\bar{x}$. We have:

$$R(x_{n,n}) = U(\alpha_m, x_{n,n}) \left(\sum_{l=0}^{q} \alpha_m^l P_l(x_{n,n}) \right) + V(\alpha_m, x_{n,n}) Q(\alpha_m)$$

and we notice that the right hand side converges to 0 when m and n grow.

Conclusion

The theory of real closed fields presented in [5] is based on an interface we now provide an instance of. A direct consequence is that real algebraic numbers immediately enjoy quantifier elimination which proves decidable its first order theory. The formalization we describe comes from various classical sources that had to be adapted, made constructive and simplified for the needs of the formalization. The methodology applied here to build algebraic numbers and make proofs feasible and quick is, up to our knowledge, original. This is also, as far as we know, the first certified formalization of real algebraic numbers in a proof assistant.

It would be interesting to provide an efficient implementation of algebraic numbers, relying on [2] and on [9] for example. The formalization we show in this paper would then serve as a reference implementation. We would need to prove the relative correctness of the efficient implementation with regard to the abstract one. But no proofs about the algebraic structure of the new implementation would be required.

It would be natural to continue this work by extending the real algebraic numbers by the imaginary unit **i**. Thanks to the constructive fundamental algebra theorem, generalized to real closed fields [4], this new field would be *algebraically closed*, *partially ordered* and would then represent the data-type of *(complex) algebraic numbers*. In the framework of Galois theory, it would also be interesting to formalize the type of algebraic extensions over rational numbers: we could then use the classical presentation and study them into their algebraic closure.

Finally, we formalized the construction of the real closure of fields of zero characteristic, which is a step in constructing the algebraic closure. It is a completely different work to formalize the algebraic closure of fields of non-zero characteristic. Moreover the efficient algorithms for the non-zero characteristic are treated in [2] and are more intricate than the ones for the zero characteristic.

Acknowledgement. I wish to thank Georges Gonthier for the numerous ideas which constitute the basis of this development and Russell O'Connor for discussions which helped me find the good way to state and prove some results. I also thank Assia Mahboubi, Enrico Tassi and the anonymous referees for their reading and comments on this paper.

References

1. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. J. of Functional Programming 13(2), 261–293 (2003); Special Issue on Logical Frameworks and Metalanguages
2. Bostan, A.: Algorithmique efficace pour des opérations de base en Calcul formel. Ph.D. thesis, École polytechnique (2003), `http://algo.inria.fr/bostan/these/These.pdf`
3. Cohen, C.: Types quotients en COQ. In: Hermann (ed.) Actes des 21éme Journées Francophones des Langages Applicatifs (JFLA 2010), INRIA, Vieux-Port La Ciotat, France (January 2010), `http://jfla.inria.fr/2010/actes/PDF/cyrilcohen.pdf`
4. Cohen, C., Coquand, T.: A constructive version of Laplace's proof on the existence of complex roots, `http://hal.inria.fr/inria-00592284/PDF/laplace.pdf` (unpublished)
5. Cohen, C., Mahboubi, A.: Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. Logical Methods in Computer Science 8(1-02), 1–40 (2012), `http://hal.inria.fr/inria-00593738`
6. Delahaye, D.: A Tactic Language for the System COQ. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)
7. 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)
8. Geuvers, H., Niqui, M.: Constructive Reals in COQ: Axioms and Categoricity. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 79–95. Springer, Heidelberg (2002), `http://dl.acm.org/citation.cfm?id=646540.696040`
9. Krebbers, R., Spitters, B.: Computer Certified Efficient Exact Reals in COQ. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 90–106. Springer, Heidelberg (2011)
10. Lang, S.: Algebra. Graduate texts in mathematics. Springer (2002)
11. Mines, R., Richman, F., Ruitenburg, W.: A course in constructive algebra. Universitext (1979); Springer-Verlag (1988)
12. 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), `http://dx.doi.org/10.1007/978-3-540-71067-7_21`
13. Project, T.M.C.: SSReflect extension and libraries, `http://www.msr-inria.inria.fr/Projects/math-components/index_html`

A Refinement-Based Approach to Computational Algebra in Coq[⋆]

Maxime Dénès[1], Anders Mörtberg[2], and Vincent Siles[2]

[1] INRIA Sophia Antipolis – Méditerranée, France
[2] University of Gothenburg, Sweden
`Maxime.Denes@inria.fr`, {`mortberg,siles`}`@chalmers.se`

Abstract. We describe a step-by-step approach to the implementation and formal verification of efficient algebraic algorithms. Formal specifications are expressed on rich data types which are suitable for deriving essential theoretical properties. These specifications are then refined to concrete implementations on more efficient data structures and linked to their abstract counterparts. We illustrate this methodology on key applications: matrix rank computation, Winograd's fast matrix product, Karatsuba's polynomial multiplication, and the gcd of multivariate polynomials.

Keywords: Formalisation of mathematics, Computer algebra, Efficient algebraic algorithms, Coq, SSReflect.

1 Introduction

In the past decade, the range of application of proof assistants has extended its traditional ground in theoretical computer science to mainstream mathematics. Formalised proofs of important theorems like the fundamental theorem of algebra [2], the four colour theorem [6] and the Jordan curve theorem [10] have advertised the use of proof assistants in mathematical activity, even in cases when the pen and paper approach was no longer tractable.

But since these results established proofs of concept, more effort has been put into designing an actually scalable library of formalised mathematics. The *Mathematical Components* project (developing the SSReflect library [8] for the Coq proof assistant) advocates the use of small scale reflection to achieve a nearly comparable level of detail to usual mathematics on paper, even for advanced theories like the proof of the Feit-Thompson theorem. In this approach, the user expresses significant deductive steps while low-level details are taken care of by small computational steps, at least when properties are decidable. Such an approach makes the proof style closer to usual mathematics.

One of the main features of these libraries is that they heavily rely on rich dependent types, which gives the opportunity to encode a lot of information

⋆ The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

directly into the type of objects: for instance, the type of matrices embeds their size, which makes operations like multiplication easy to implement. Also, algorithms on these objects are simple enough so that their correctness can easily be derived from the definition. However in practice, most efficient algorithms in modern computer algebra systems do not rely on dependent types and do not provide any proof of correctness. We show in this paper how to use this rich mathematical framework to develop efficient computer algebra programs *with proofs of correctness*. This is a step towards closing the gap between proof assistants and computer algebra systems.

The methodology we suggest for achieving this is the following: we are able to prove the correctness of some mathematical algorithms having all the high-level theory at our disposal and we then refine them to an implementation on simpler data structures that will be actually running on machines. In short, we aim at formally linking convenient high-level properties to efficient low-level implementations, ensuring safety of the whole approach while enjoying better performance thanks to the separation of proofs and computational content.

In the next section, we describe the methodology of refinements. Then, we give two examples of such refinements for matrices in Section 3, and polynomials in Section 4. In Section 5, we give a solution to unify both examples by describing CoqEAL[1], a library built using this methodology on top of the SSReflect libraries.

2 Refinements

Refinements are commonly used to describe successive steps when verifying a program. Typically, a specification is expressed in Hoare logic, then the program is described in a high-level language and finally implemented in C. Each step is proved correct with respect to the previous one. By using several formalisms, one has to trust every translation step or prove them correct in yet another formalism.

Our approach is similar: we refine the definition of a concept to an efficient algorithm described on high-level data structures. Then, we implement it on data structures that are closer to machine representations, once we no longer need rich theory to prove the correctness. Thus the implementation is an immediate translation of the algorithm, see Fig. 1.

However, in our approach, the three layers can be expressed in the same formalism (the Calculus of Inductive Constructions), though they do not use exactly the same features. On one hand, the high-level layers use rich dependent types that are very useful when describing theories because they allow abuse of notations and concise statements which quickly become necessary when working with advanced mathematics. On the other hand, the efficient implementations use simple types, which are closer to standard implementations in traditional

[1] Documentation available at `http://www-sop.inria.fr/members/Maxime.Denes/coqeal/`

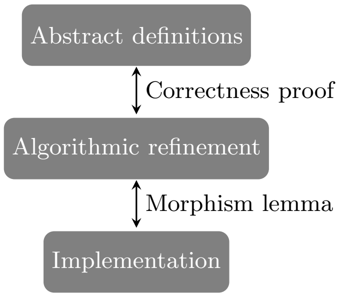


Fig. 1. The three steps of refinement

programming languages. The main advantage of this approach is that the correctness of translations can easily be expressed in the formalism itself, and we do not rely on any additional external proofs.

In the next sections, we are going to use the following methodology to build efficient algorithms from high-level descriptions:

1. Implement an abstract version of the algorithm using SSReflect's structures and use the libraries to prove properties about them. Here we can use the full power of dependent types when proving correctness.
2. Refine this algorithm into an efficient one using SSReflect's structures and prove that it behaves like the abstract version.
3. Translate the SSReflect structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

3 Matrices

Linear algebra is a natural first test-case to validate our approach, as a pervasive and inherently computational area of mathematics, which is well covered by the SSReflect library [7]. In this section, we will detail the (quite simple) data structure we use to represent matrices and then review two fundamental examples: rank computation and efficient matrix product.

3.1 Representation

Matrices are represented by finite functions over pairs of ordinals (the indices):

```
(* 'I_n *)
Inductive ordinal (n : nat) : predArgType := Ordinal m of m < n.

(* 'M[R]_(m,n) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

This encoding makes many properties easy to derive, but it is inefficient for evaluation. Indeed, a finite function over `'I_m * 'I_n` is internally represented as a flat list of `m` × `n` values which has to be traversed whenever the function is evaluated. Moreover, having the size of matrices encoded in their type allows to state concise lemmas without explicit side conditions, but it is not always flexible enough when getting closer to machine-level implementation details.

To be able to implement efficient matrix operations we introduce a low-level data type `seqmatrix` representing matrices as lists of lists. A concrete matrix is built from an abstract one by mapping canonical enumerations (`enum`) of ordinals to the corresponding coefficients in the abstract matrix:

```
Definition seqmx_of_mx (M : 'M[R]_(m,n)) : seqmatrix :=
  [seq [seq M i j | j <- enum 'I_n] | i <- enum 'I_m].
```

To ensure the correct behaviour of concrete matrices it is sufficient to prove that `seqmx_of_mx` is injective (`==` denotes boolean equality):

```
Lemma seqmx_eqP (M N : 'M[R]_(m,n)) :
  reflect (M = N) (seqmx_of_mx M == seqmx_of_mx N).
```

Operations like addition are straightforward to implement, and their correctness is expressed through a morphism lemma, stating that the concrete representation of the sum of two matrices is the concrete sum of their concrete representations:

```
Definition addseqmx (M N : seqmatrix) : seqmatrix :=
  zipwith (zipwith (fun x y => add x y)) M N.

Lemma addseqmxE :
  {morph (@seqmx_of_mx m n) : M N / M + N >-> addseqmx M N}.
```

Here `morph` is notation meaning that `seqmx_of_mx` is an additive morphism from abstract to concrete matrices. It is worth noting that we could have stated all our morphism lemmas with the converse operator (from concrete matrices to abstract ones). But these lemmas would then have been quantified over lists of lists, with poorer types, which would have required a well-formedness predicate as well as premises expressing size constraints. The way we have chosen takes full advantage of the information carried by richer types.

Like the `addseqmx` operation, we have developed concrete implementations of most of the matrix operations provided by the SSReflect library and proved the corresponding morphism lemmas. Among these operations we can cite: subtraction, scaling, transpose and block operations.

3.2 Computing the Rank

Now that the basic data structure and operations have been defined, it is possible to apply our approach to an algorithm based on Gaussian elimination which computes the rank of a matrix $A = (a_{i,j})$ over a field K. We first specify the algorithm using abstract matrices and then refine it to the low-level structures.

An elimination step consists of finding a nonzero pivot in the first column of A. If there is none, it is possible to drop the first column without changing the rank. Otherwise, there is an index i such that $a_{i,1} \neq 0$. By linear combinations of rows (preserving the rank) A can be transformed into the following matrix B:

$$B = \begin{bmatrix} 0 & a_{1,2} - \frac{a_{1,1} \times a_{i,2}}{a_{i,1}} & \cdots & a_{1,n} - \frac{a_{1,1} \times a_{i,n}}{a_{i,1}} \\ & \vdots & & \vdots \\ 0 & & & \\ a_{i,1} & a_{i,2} & \cdots & a_{i,n} \\ 0 & \vdots & & \vdots \\ 0 & a_{n,2} - \frac{a_{n,1} \times a_{i,2}}{a_{i,1}} & \cdots & a_{n,n} - \frac{a_{n,1} \times a_{i,n}}{a_{i,1}} \end{bmatrix} = \begin{bmatrix} 0 & \\ \vdots & \boxed{R_1} \\ 0 & \\ a_{i,1} & \cdots \quad a_{i,n} \\ 0 & \\ \vdots & \boxed{R_2} \\ 0 & \end{bmatrix}$$

Now pose $R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$, since $a_{i,1} \neq 0$, this means that $rank\ A = rank\ B = 1 + rank\ R$. Hence the current rank can be incremented and the algorithm can be recursively applied on R.

In our development we defined a function `elim_step` returning the matrix R above and a boolean b indicating if a pivot has been found. A wrapper function `rank_elim` is in charge of maintaining the current rank and performing the recursive call on R:

```
Fixpoint rank_elim (m n : nat) {struct n} : 'M[K]_(m,n) -> nat :=
  match n return 'M[K]_(m,n) -> nat with
   | q.+1 => fun M =>
     let (R,b) := elim_step M in (rank_elim R + b)%N
   | _ => fun _ => 0%N
  end.
```

Note that booleans are coerced to natural numbers: b is interpreted as 1 if true and 0 if false. The correctness of `rank_elim` is expressed by relating it to the `\rank` function of the SSReflect library:

```
Lemma rank_elimP n m (M : 'M[K]_(m,n)) : rank_elim M = \rank M.
```

The proof of this specification relies on a key invariant of `elim_step`, relating the ranks of the input and output matrices:

```
Lemma elim_step_rank m n (M : 'M[K]_(m, 1 + n)) :
  let (R,b) := elim_step M in \rank M = (\rank R + b)%N.
```

Now the proof of `rank_elimP` follows by induction on n. The concrete version of this algorithm is a direct translation of the algorithm using only concrete matrices and executable operations on them. This executable version (called `rank_elim_seqmx`) is then linked to the abstract implementation by the lemma:

```
Lemma rank_elim_seqmxE : forall m n (M : 'M[K]_(m, n)),
  rank_elim_seqmx m n (seqmx_of_mx M) = rank_elim M.
```

The proof of this is straightforward as all of the operations on concrete matrices have morphism lemmas which means that the proof can be done simply by expanding the definitions and applying the translation morphisms.

3.3 Fast Matrix Product

In the context we presented, the naïve matrix product (i.e. with cubic complexity) of two matrices M and N can be implemented by transposing the list of lists representing N and then for each i and j compute $\sum_k M_{i,k} N^{\mathrm{T}}_{j,k}$:

```
Definition mulseqmx (M N : seqmatrix) : seqmatrix :=
  let N' := trseqmx N in
  map (fun r => map (foldl2 (fun z x y => x * y + z) 0 r) N') M.

Lemma mulseqmxE (M : 'M[R]_(m,p)) (N : 'M[R]_(p,n)) :
  mulseqmx (seqmx_of_mx M) (seqmx_of_mx N) = seqmx_of_mx (M *m N).
```

`*m` is SSReflect's notation for the matrix product. Once again, the rich type information in the quantification of the morphism lemma ensures that it can be applied only if the two matrices have compatible sizes.

In 1969, Strassen [19] showed that 2×2 matrices can be multiplied using only 7 multiplications without requiring commutativity. This yields an immediate recursive scheme for the product of two $n \times n$ matrices with $\mathcal{O}(n^{\log_2 7})$ complexity.[2] This is an important theoretical result, since matrix multiplication was commonly thought to be intrinsically of cubic complexity, it opened the way to many further improvements and gave birth to a fertile branch of algebraic complexity theory.

However, Strassen's result is also still of practical interest since the asymptotically best algorithms known today [4] are slower in practice because of huge hidden constants. Thus, we implemented a variant of this algorithm suggested by Winograd in 1971 [20], decreasing the required number of additions and subtractions to 15 (instead of 18 in Strassen's original proposal). This choice reflects the implementation of matrix product in most of modern computer algebra systems. A previous formal description of this algorithm has been developed in ACL2 [17], but it is restricted to matrices whose sizes are powers of 2. The extension to arbitrary matrices represents a significant part of our development, which is to the best of our knowledge the first complete formally verified description of Winograd's algorithm.

We define a function expressing a recursion step in Winograd's algorithm. Given two matrices A and B and an operator f representing matrix product, it reformulates the algebraic identities involved in the description of the algorithm:

```
Definition winograd_step {p : positive} (A B : 'M[R]_(p + p)) f :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
```

[2] $\log_2 7$ is approximately 2.807.

```
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in let Y := B22 - B12 in
  let C21 := f X Y in
  let X := A21 + A22 in let Y := B12 - B11 in
  let C22 := f X Y in
  let X := X - A11 in let Y := B22 - Y in
  let C12 := f X Y in
  let X := A12 - X in
  let C11 := f X B22 in
  let X := f A11 B11 in
  let C12 := X + C12 in let C21 := C12 + C21 in
  let C12 := C12 + C22 in let C22 := C21 + C22 in
  let C12 := C12 + C11 in
  let Y := Y - B21 in
  let C11 := f A22 Y in let C21 := C21 - C11 in
  let C11 := f A12 B21 in let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

This is an implementation of matrix multiplication that is clearly not suited for proving algebraic properties, like associativity. The correctness of this function is expressed by the fact that if f is instantiated by the multiplication of matrices, `winograd_step A B` should be the product of `A` and `B` (`=2` denotes extensional equality):

```
Lemma winograd_stepP (p : positive) (A B : 'M[R]_(p + p)) f :
  f =2 mulmx -> winograd_step A B f = A *m B.
```

This proof is made easy by the use of the `ring` tactic (the script is two lines long). Since version 8.4 of Coq, `ring` is applicable to non-commutative rings, which has allowed its use in our context.

Note that the above implementation only works for even-sized matrices. This means that the general procedure has to implement a strategy for handling odd-sized matrices. Several standard techniques have been proposed, which fall into two categories. Some are static, in the sense that they preprocess the matrices to obtain sizes that are powers of 2. Others are dynamic, meaning that parity is tested at each recursive step. Two standard treatments can be implemented either statically or dynamically: padding and peeling. The first consists of adding rows and/or columns of zeros as required to get even dimensions (or a power of 2), these lines are then simply removed from the result. Peeling on the other hand removes rows or columns when needed, and corrects the result accordingly.

We chose to implement dynamic peeling because it seemed to be the most challenging technique from the formalisation point of view, since the size of matrices involved depend on dynamic information and the post processing of the result is more sophisticated than using padding. Another motivation is that dynamic peeling has shown to give good results in practice.

The function that implements Winograd multiplication with dynamic peeling is called `winograd` and it is proved correct with respect to the usual matrix product:

```
Lemma winogradP : forall (n : positive) (M N : 'M[R]_n),
  winograd M N = M *m N.
```

The concrete version is called `winograd_seqmx` and it is also just a direct translation of `winograd` using only concrete operations on `seq` based matrices. In the next section, Fig. 2 shows some benchmarks of how well this implementation performs compared to the naïve matrix product, but we will first discuss how to implement concrete algorithms based on dependently typed polynomials.

4 Polynomials

Polynomials in the SSReflect library are represented as records with a list representing the coefficients and a proof that the last of these is nonzero. The library also contains basic operations on this representation like addition and multiplication and proofs that the polynomials form a commutative ring using these operations. The implementation of these operations use big operators [3] which means that it is not possible to compute with them.

To remedy this we have implemented polynomials as lists without any proofs together with executable implementations of the basic operations. It is very easy to build a concrete polynomial from an abstract polynomial, simply apply the record projection (called `polyseq`) to extract the list from the record. The soundness of concrete polynomials is proved by showing that the pointwise boolean equality on the projected lists reflects the equality on abstract polynomials:

```
Lemma polyseqP p q : reflect (p = q) (polyseq p == polyseq q).
```

Basic operations like addition and multiplication are slightly more complicated to implement for concrete polynomials than for concrete matrices as it is necessary to ensure that these operations preserve the invariant that the last element is nonzero. For instance multiplication is implemented as:

```
Fixpoint mul_seq p q := match p,q with
  | [::], _ => [::]
  | _, [::] => [::]
  | x :: xs,_ => add_seq (scale_seq x q) (mul_seq xs (0%R :: q))
  end.

Lemma mul_seqE : {morph polyseq : p q / p * q >-> mul_seq p q}.
```

Here `add_seq` is addition of concrete polynomials and `scale_seq x q` means that every coefficient of `q` is multiplied by `x` (both of these are implemented in such a way that the invariant that the last element is nonzero is satisfied). Using this approach we have implemented a substantial part of the SSReflect polynomial library, including pseudo-division, using executable polynomials.

4.1 Fast Polynomial Multiplication

The naïve polynomial multiplication algorithm presented in the previous section requires $\mathcal{O}(n^2)$ operations. A more efficient algorithm is Karatsuba's algorithm [1,11] which is a divide and conquer algorithm based on reducing the number of recursive calls in the multiplication. More precisely, in order to multiply two polynomials written as $aX^k + b$ and $cX^k + d$ the ordinary method

$$(aX^k + b)(cX^k + d) = acX^{2k} + (ad + bc)X^k + cd$$

requires four multiplications (as the multiplications by X^n can be implemented efficiently by padding the list of coefficients by n zeroes). The key observation is that this can be rewritten as

$$(aX^k + b)(cX^k + d) = acX^{2k} + ((a + b)(c + d) - ac - bd)X^k + bd$$

which only requires three multiplication: ac, $(a+b)(c+d)$ and bd. Now if the two polynomials have 2^n coefficients and the splitting is performed in the middle at every point then the algorithm will only require $\mathcal{O}(n^{\log_2 3})$ which is better than the naïve algorithm.[3] If the polynomials do not have 2^n coefficients it is possible to split the polynomials at for example $\lfloor n/2 \rfloor$ as the formula above holds for any $k \in \mathbb{N}$ and still obtain a faster algorithm. This algorithm has been implemented in Coq previously for binary natural numbers [15] and for numbers represented by a tree-like structure [9]. But as far as we know, it has never been implemented for polynomials before. When implementing this algorithm we first implemented it using dependently typed polynomials as:

```
Fixpoint karatsuba_rec (n : nat) p q := match n with
  | 0%N => p * q
  | n'.+1 => if (size p <= 2) || (size q <= 2) then p * q else
     let m := minn (size p)./2 (size q)./2 in
     let (p1,p2) := splitp m p in
     let (q1,q2) := splitp m q in
     let p1q1 := karatsuba_rec n' p1 q1 in
     let p2q2 := karatsuba_rec n' p2 q2 in
     let p12 := p1 + p2 in
     let q12 := q1 + q2 in
     let p12q12 := karatsuba_rec n' p12 q12 in
     p1q1 * 'X^(2 * m) + (p12q12 - p1q1 - p2q2) * 'X^m + p2q2
  end.
```

Here `splitp` is a function that splits the polynomial at the correct point using `take` and `drop`. There is also a wrapper function named `karatsuba` that calls `karatsuba_seq` with the greatest degree of `p` and `q`. The correctness of this algorithm is expressed by:

```
Lemma karatsubaE : forall p q, karatsuba p q = p * q.
```

[3] $\log_2 3$ is approximately 1.585.

As p and q are SSReflect polynomials this lemma can be proved using all of the theory in the library. The next step is to implement the executable version (karatsuba_seq) of this algorithm which is done by changing all the operations in the above version to executable operations on concrete polynomials. The correctness of the concrete algorithm is then proved by:

```
Lemma karatsuba_seqE :
  {morph polyseq : p q / karatsuba p q >-> karatsuba_seq p q}.
```

The proof of this is straightforward as all of the operations have morphism lemmas for translating back and forth between the concrete representation and the high-level ones.

In Fig. 2 the running time of the different multiplication algorithms that we have implemented is compared:

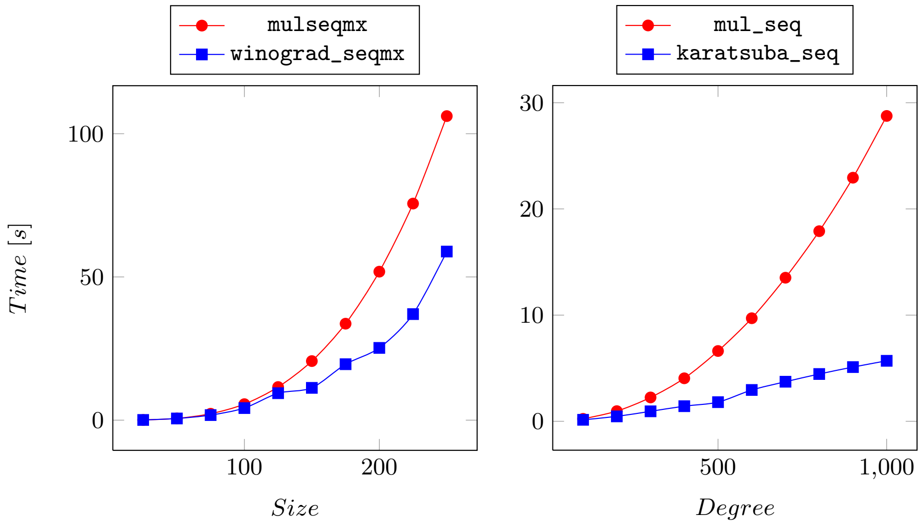


Fig. 2. Benchmarks of Winograd and Karatsuba multiplication

The benchmarks have been done by computing the square of integer matrices and polynomials using the Coq virtual machine (i.e. by running vm_compute). It is clear that both the implementation of Winograd matrix multiplication and Karatsuba polynomial multiplication is faster than their naïve counterparts, as expected.

4.2 GCD of Multivariate Polynomials

An important feature of modern computer algebra systems is to compute the greatest common divisor (gcd) of multivariate polynomials. The main idea of our implementation is based on the observation that in order to compute the gcd of elements in $R[X_1, \ldots, X_n]$ it suffices to show how to compute the gcd in $R[X]$

given that it is possible to compute the gcd of elements in R. So to compute the gcd of elements in $\mathbb{Z}[X, Y]$ we model it as $(\mathbb{Z}[X])[Y]$, i.e. univariate polynomials in Y with coefficients in $\mathbb{Z}[X]$, and then use that there is a gcd algorithm in $\mathbb{Z}$.

The algorithm that we implemented is based on the presentation of Knuth in [12] which uses that in order to compute the gcd of two multivariate polynomials it is possible to instead consider the task of computing the gcd of *primitive* polynomials, i.e. polynomials where all coefficients are coprime. Using that any polynomial can be split in a primitive part and a non-primitive part by dividing by the gcd of its coefficients (this is called the *content* of the polynomial) we get an algorithm for computing the gcd of any two polynomials. Below is our implementation of this algorithm together with explanations of the operations:

```
Fixpoint gcdp_rec (n : nat) (p q : {poly R}) :=
  let r := modp p q in
  if r == 0 then q
            else if n is m.+1 then gcdp_rec m q (pp r) else pp r.

Definition gcdp p q :=
  let (p1,q1) := if size p < size q then (q,p) else (p,q) in
  let d := (gcdr (gcdsr p1) (gcdsr q1))%:P in
  d * gcdp_rec (size (pp p1)) (pp p1) (pp q1).
```

– `modp p q` computes the remainder after pseudo-dividing `p` by `q`.
– `pp p` computes the primitive part of `p` by dividing it by its content.
– `gcdsr p` computes the content of `p`.
– `gcdr (gcdsr p1)(gcdsr q1)` computes the gcd (using the operation in the underlying ring) of the content of `p1` and the content of `q1`.

The correctness of this algorithm is now expressed by:

```
Lemma gcdpP : forall p q g, g %| gcdp p q = (g %| p) && (g %| q).
```

Here `p %| q` computes whether `p` divides `q` or not. As divisibility is reflexive this equality is a compact way of expressing that the function actually computes the gcd of `p` and `q`.

Our result is stated in constructive algebra [14] as: If R is a gcd domain then so is $R[X]$. Our algorithmic proof is different (and arguably simpler) than the one in [14]; for instance, we do not go via the field of fractions of the ring.

As noted in [12], this algorithm may be inefficient when applied on the polynomials over integers. The reference [12] provides a solution in this case, based on subresultants. This would be a further refinement of the algorithm, which would be interesting to explore since subresultants have been already analysed in Coq [13].

The executable version (`gcdp_seq`) of the algorithm has also been implemented and is linked to the abstract version above by:

```
Lemma gcdp_seqE :
  {morph polyseq : p q / gcdp p q >-> gcdp_seq p q}.
```

But when running the concrete implementation there is a quite subtle problem: the `polyseq` projection links the abstract polynomials with the concrete polynomials of type `seq R` where `R` is a *ring* with a gcd operation. Let us consider multivariate polynomials, for example $R[x, y]$. In this case the concrete type will be `seq (seq R)`, but `seq R` is not a ring so our algorithm is not applicable! The next section explains how to resolve this issue so that it is possible to implement computable algorithms of the above kind that rely on the computability of the underlying ring.

5 Algebraic Hierarchy Of Computable Structures

As noted in the previous section there is a problem when implementing multivariate polynomials by iterating the polynomial construction, i.e. by representing $R[X, Y]$ as $(R[X])[Y]$. The same problem occurs when considering other structures where the computation relies on the computability of the underlying ring as is the case when computing the characteristic polynomial of a square matrix for instance. For this, one needs to compute with matrices of polynomials which will require a concrete implementation of matrices with coefficients being a concrete implementation of polynomials.

However, both the list based matrices and polynomials have something in common: we can guarantee the correctness of the operations on a subset of the low-level structure. This can be used to implement another hierarchy of computable structures corresponding to the SSReflect algebraic hierarchy.

5.1 Design of the Library

We have implemented computable counterparts to the basic structures in this hierarchy, e.g. $\mathbb{Z}$-modules, rings and fields. These are implemented in the same manner as presented in [5] using canonical structures. Here are a few examples of the mixins we use:

```
Record trans_struct (A B: Type) : Type := Trans {
  trans : A -> B;
  _ : injective trans
}.

(* Mixin for "Computable" Z-modules *)
Record mixin_of (V : zmodType) (T: Type) : Type := Mixin {
  zero : T;
  opp : T -> T;
  add : T -> T -> T;
  tstruct : trans_struct V T;
  _ : (trans tstruct) 0 = zero;
  _ : {morph (trans tstruct) : x / - x >-> opp x};
  _ : {morph (trans tstruct) : x y / x + y >-> add x y}
}.
```

```
(* Mixin for "Computable" Rings *)
Record mixin_of (R : ringType) (V : czmodType R) : Type := Mixin {
  one : V;
  mul : V -> V -> V;
  _ : (trans V) 1 = one;
  _ : {morph (trans V) : x y / x * y >-> mul x y}
}.
```

The type `czmodType` is the computable $\mathbb{Z}$-module type parametrized by a $\mathbb{Z}$-module. The `trans` function is the translation function from SSReflect structures to the computable structures and the only property that is required of it is that it is injective, so we are sure that different high-level objects are mapped to different computable objects.

This way we can implement all the basic operations of the algebraic structures the way we want (for example using fast matrix multiplication as an implementation of `*m` instead of a naïve one), and the only thing we have to prove is that the implementations behave the same as SSReflect's operations *on the subset of "well-formed terms"* (e.g. for polynomials, lists that do not end with 0). This is done by providing the corresponding morphism lemmas.

The operations presented in the previous sections can then be implemented by having computable structures as the underlying structure instead of dependently typed ones. This way one can prove that polynomials represented as lists is a computable ring by assuming that the coefficients are computable and hence get ring operations that can be applied on multivariate polynomials built by iterating the construction.

It is interesting to note that the equational behavior of an abstract structure is carried as a parameter, but does not appear in its computable counterpart, which depends only on the operations to be implemented. For instance, the same computable ring structure can implement a commutative ring or an arbitrary one, only its parameter varies.

5.2 Example: Computable Ring of Polynomials

Let us explain how the list based polynomials can be made a computable ring. First, we define:

```
Variable R : comRingType.
Variable CR : cringType R.
```

This says that `CR` is a computable ring parametrized by a commutative ring which makes sense as any commutative ring is a ring. Next we need to implement the translation function from `{poly R}` to `seq CR` and prove that this translation is injective:

```
Definition trans_poly (p : {poly R}) : seq CR :=
  map (@trans R CR) (polyseq p).

Lemma inj_trans_poly : injective trans_poly.
```

Assuming that computable polynomials already are an instance of the computable $\mathbb{Z}$-module structure it is possible to prove that they are computable rings by implementing multiplication (exactly like above) and then prove the corresponding morphism lemmas:

```
Lemma trans_poly1 : trans_poly 1 = [:: (one CR)].

Lemma mul_seqE :
  {morph trans_poly : p q / p * q >-> mul_seq p q}.
```

At this point, we could also have used the `karatsuba_seq` implementation of polynomial multiplication instead of `mul_seq` since we can prove its correctness using the `karatsubaE` and `karatsuba_seqE` lemmas. Finally this can be used to build the `CRing` mixin and make it a canonical structure.

```
Definition seq_cringMixin := CRingMixin trans_poly1 mul_seqE.

Canonical Structure seq_cringType :=
  Eval hnf in CRingType {poly R} seq_cringMixin.
```

5.3 Examples of Computations

This computable ring structure has also been instantiated by the Coq implementation of $\mathbb{Z}$ and $\mathbb{Q}$ which means that they can be used as basis when building multivariate polynomials. To multiply $2 + xy$ and $1 + x + xy + x^2y^2$ in $\mathbb{Z}[x, y]$ one can write:

```
Definition p := [:: [:: 2]; [:: 0; 1]].
Definition q := [:: [:: 1; 1]; [:: 0; 1]; [:: 0; 0; 1]].

> Eval compute in mul p q.
  = [:: [:: 2; 2]; [:: 0; 3; 1]; [:: 0; 0; 3]; [:: 0; 0; 0; 1]]
```

The result should be interpreted as $(2 + 2x) + (3x + x^2)y + 3x^2y^2 + x^3y^3$. The gcd of $1 + x + (x + x^2)y$ and $1 + (1 + x)y + xy^2$ in $\mathbb{Z}[x, y]$ can be computed by:

```
Definition p := [:: [:: 1; 1] ; [:: 0; 1; 1] ].
Definition q := [:: [:: 1]; [:: 1; 1]; [:: 0; 1]].

> Eval compute in gcdp_seq p q.
  = [:: [:: 1]; [:: 0; 1]]
```

The result is $1 + xy$ as expected. The following is an example over $\mathbb{Q}[x, y]$:

```
Definition p := [:: [:: 2 # 3; 2 # 3]; [:: 0; 1 # 2; 1 # 2]].
Definition q := [:: [:: 2 # 3]; [:: 2 # 3; 1 # 2]; [:: 0; 1 # 2]].

> Eval compute in gcdp_seq p q.
  = [:: [:: 1 # 3]; [:: 0; 1 # 4]]
```

The two polynomials are $\frac{2}{3} + \frac{2}{3}x + \frac{1}{2}xy + \frac{1}{2}x^2y$ and $\frac{2}{3} + \frac{2}{3}y + \frac{1}{2}xy + \frac{1}{2}xy^2$. The resulting gcd should be interpreted as $\frac{1}{3} + \frac{1}{4}xy$.

6 Conclusions and Further Work

In this paper, we showed how to use high-level libraries to prove properties of algorithms, while retaining good execution capabilities by providing efficient low-level implementations. The need of modularity of the executable structure appears naturally and the methodology explained in [5] works quite well. The only thing a user has to provide is a proof of an injectivity lemma stating that the translation behaves correctly.

The methodology we suggest has already been used in other contexts, like the CoRN library, where properties of real numbers described in [16] are obtained by proving that these real numbers are isomorphic to an abstract, pre-existing but less efficient version. We tried to show that this approach can be applied in a systematic and modular way.

The library we designed also helps to solve a restriction of SSReflect: due to a lot of computations during deduction steps, some of the structures are *locked* to allow type-checking to be performed in a reasonable amount of time. This locking prevents full-scale reflection on some of the most complex types like big operators, polynomials or matrices. Our implementation restores the ability to perform full-scale reflection on abstract structures, and more generally to compute. For instance, addition of two fully instantiated polynomials cannot be evaluated to its actual numerical result but we can refine it to a computable object that will reduce. This is a first step towards having in the same system definitions of objects on which properties can be proved and some of the usual features of a computer algebra system.

However, in its current state, the inner structure of our library is slightly more rigid than necessary: we create a type for computable $\mathbb{Z}$-modules, but in practice, all the operations it contains could be packaged independently. Indeed, on each of these operations we prove only a morphism lemma linking it to its abstract counterpart, whereas in usual algebraic structures, expressing properties like distributivity require access to several operations at once. This specificity would make it possible to reorganise the library and create independent structures for each operation, instead of creating one of them for each type. Also, we could use other packaging methods, like type classes [18], to simplify the layout of the library. However, modifying the library to use type classes on top of SSReflect's canonical structures is still on-going work, since we faced some incompatibilities between the different instance resolution mechanisms.

References

1. Abdeljaoued, J., Lombardi, H.: Méthodes matricielles - Introduction à la complexité algébrique. Springer (2004)
2. Barendregt, H., Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: The "fundamental theorem of algebra" project, http://www.cs.ru.nl/~freek/fta/
3. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical Big Operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 86–101. Springer, Heidelberg (2008)

4. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9(3), 251–280 (1990)
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. Gonthier, G.: Formal proof—the four-color theorem. In: Notices of the American Mathematical Society, vol. 55, pp. 1382–1393 (2008)
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.: A Small Scale Reflection Extension for the COQ system. Technical report, Microsoft Research INRIA (2009), http://hal.inria.fr/inria-00258384
9. Grégoire, B., Théry, L.: A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 423–437. Springer, Heidelberg (2006)
10. Hales, T.C.: The jordan curve theorem, formally and informally. In: The American Mathematical Monthly, vol. 114, pp. 882–894 (2007)
11. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. In: USSR Academy of Sciences, vol. 145, pp. 293–294 (1962)
12. Knuth, D.E.: The art of computer programming, vol.2: seminumerical algorithms. Addison-Wesley (1981)
13. Mahboubi, A.: Proving Formally the Implementation of an Efficient gcd Algorithm for Polynomials. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 438–452. Springer, Heidelberg (2006)
14. Mines, R., Richman, F., Ruitenburg, W.: A Course in Constructive Algebra. Springer (1988)
15. O'Connor, R.: Karatsuba's multiplication, http://coq.inria.fr/V8.2pl1/contribs/Karatsuba.html
16. 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)
17. Palomo-Lozano, F., Medina-Bulo, I., Alonso-Jiménez, J.: Certification of matrix multiplication algorithms. strassen's algorithm in ACL2. In: Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, Informatics Research Report EDI-INF-RR-0046, Edinburgh (2001)
18. 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)
19. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13(4), 354–356 (1969)
20. Winograd, S.: On multiplication of 2x2 matrices. Linear Algebra and its Applications 4, 381–388 (1971)

Bridging the Gap: Automatic Verified Abstraction of C

David Greenaway[1,2], June Andronick[1,2], and Gerwin Klein[1,2]

[1] NICTA, Sydney, Australia[⋆]
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{David.Greenaway,June.Andronick,Gerwin.Klein}@nicta.com.au

Abstract. Before low-level imperative code can be reasoned about in an interactive theorem prover, it must first be converted into a logical representation in that theorem prover. Accurate translations of such code should be conservative, choosing safe representations over representations convenient to reason about. This paper bridges the gap between conservative representation and convenient reasoning. We present a tool that automatically abstracts low-level C semantics into higher level specifications, while generating proofs of refinement in Isabelle/HOL for each translation step. The aim is to generate a verified, human-readable specification, convenient for further reasoning.

1 Introduction

Low-level imperative C is still the most widely used language for developing software with high performance and precise memory requirements, especially in embedded and critical high-assurance systems. The challenge of formally verifying C programs has been attacked with approaches ranging from static analysis for eliminating certain runtime errors to full functional correctness with respect to a high-level specification. This paper addresses the latter by improving automation in the verification process while preserving the strength of the correctness proof.

The first step required to formally reason about a program is to parse the code into a formal logic. The parser is necessarily trusted, giving rise to two approaches: either the parser is kept simple, minimising the assumption we make about its correctness, but resulting in a low-level formal model; or the parser generates a specification that is more pleasant to reason about, but resulting in a weaker trust chain.

We advocate the first approach, increasing the level of trustworthiness of the final proof. In this context, existing solutions either bear the burden of working with the low level C semantics, as for instance in Verisoft [2], or manually bridge the gap by abstracting the low-level specification to an intermediate functional specification as in the L4.verified project [7], which showed correctness of the seL4 microkernel.

[⋆] 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.

```c
int max(int a, int b) {
  if (a < b)
    return b;
  return a;
}

int gcd(int a, int b) {
  int c;
  while (a != 0) {
    c = a;
    a = b % a;
    b = c;
  }
  return b;
}
```

```
max $a$ $b$ $\equiv$
  if $a <_s b$ then $b$ else $a$

gcd $a$ $b$ $\equiv$
  do
    $(a, b) \leftarrow$ while $(\lambda(a, b)\ s.\ a \neq 0)$
        $(\lambda(a, b).$ return $(b$ mod $a, a))$
        $(a, b)$;
    return $b$
  od
```

Fig. 1. C functions `max` and `gcd` and their corresponding abstractions

The contribution of this paper is a new tool[1] that automatically abstracts low-level C semantics into higher level specifications, while generating proofs in Isabelle/HOL for each translation step. The aim is to generate a human-readable specification that is easier and more convenient to reason about than the original code. Simpler specifications are more amenable to proving further high-level properties: instead of 25 person years reasoning on the code level, establishing integrity and authority confinement for seL4 merely took 10 person months, because it could be proved about the much simpler, abstract specification instead (with the functional correctness proof guaranteeing that it is then true down to the C code level). One third of these 25 person years were dedicated to refinement proofs of the form we envision our tool to eventually automate. The novelty here lies in providing both automated abstraction and correctness proofs.

As a running example, we will consider two simple functions, computing the maximum and the greatest common divisor respectively of two numbers. The C implementation of these two functions is given in Fig 1 on the left and the translation output of our tool on the right. In comparison, Fig 2 shows the output of the L4.verified C parser by Norrish [14,11] in the Simpl language [12] embedded in Isabelle/HOL [10]. The output of this parser is the starting point of our translation.

Compared to the C parser output, the result of our tool is significantly simpler and more abstract. The raw parser output is so complex because the C semantics have to deal with abrupt termination (e.g., `return` statements), with ensuring the C standard is obeyed (*guard* statements), with non-terminating loops, etc. Modelling these conservatively and precisely with a minimal trusted computing base induces overhead. Our tool aims to automatically distill the interesting semantic content without sacrificing trust.

While the above are toy examples for presentation, the tool is not: it successfully translates, for instance, the seL4 microkernel with ca. 8 700 lines of code, a malloc-style allocator, and a real-time operating system task scheduler. Where insightful, we will mention results of applying the tool to these code bases.

[1] Available at `http://ssrg.nicta.com.au/projects/TS/autocorres/`

```
TRY
    IF {| ´a <_s ´b|} THEN
        ´ret-int :== ´b;
        ´exn-var :== Return;
        THROW
    ELSE
        SKIP
    FI;
    ´ret-int :== ´a;
    ´exn-var :== Return;
    THROW;
    GUARD DontReach ∅
        SKIP
CATCH
    SKIP
END
```

(a) `max` Simpl Translation

```
TRY
    NonDetInit c-' c-'-update;
    WHILE {| ´a ≠ 0|} DO
        ´c :== ´a;
        GUARD Div-0 {| ´a ≠ 0|}
            ´a :== ´b mod ´a;
        ´b :== ´c
    OD;
    ´ret-int :== ´b;
    ´exn-var :== Return;
    THROW;
    GUARD DontReach ∅
        SKIP
CATCH
    SKIP
END
```

(b) `gcd` Simpl Translation

Fig. 2. The C functions from Fig 1 parsed into Simpl

Current limitations of the tool are: recursion is not supported, and a limited number of features of the C language are not supported, most notably taking the address of local variables. The first limitation is planned for future work, while the second limitation stems from the C parser front-end.

In the following, Sec 2 describes the supported C subset, the input language Simpl and the monadic framework the tool is working in. Sec 3 presents the core of the tool by explaining the translations in the abstraction process and their proofs, while Sec 3.7 describes the final theorem between tool input and output.

2 Background

2.1 Parsing C

Before code can be reasoned about, it must first be translated into the theorem prover. In this work, we consider programs in C99 [6] translated into Isabelle/HOL using Norrish's C parser [14,11]. This parser supports a subset of C, including loops, function calls, type casting, pointer arithmetic and structures. Integer arithmetic is defined to match a two's-complement 32-bit system. The parser emits inline *guards* to ensure that undefined operations, such as divide-by-zero or signed integer overflow, do not occur. As the parser must be trusted, it attempts to be simple, giving the most literal translation of C wherever possible.

The parser does not support `goto` statements, expressions with side-effects, references to local variables, `switch` statements using fall-through, unions, floating point arithmetic, or calls to function pointers. Finally, while the parser does support recursion, our tool does not yet handle such inputs. Our tool remains useful despite these limitations as embedded and systems code is often stack depth-constrained and typically avoids recursion.

2.2 Simpl

The parser translates C source code into Schirmer's *Simpl* language [12] embedded in Isabelle/HOL. Simpl is a generic imperative language with deeply embedded statements and shallowly embedded expressions, designed to be a target for embedding programs in a variety of languages, such as C, Java and Ada.

The Simpl language consists of 11 commands. The commands of interest are:

$$\begin{array}{l} c \equiv \mathsf{SKIP} \mid \mathsf{BASIC}\ m \mid c_1\ ;\ c_2 \mid \mathsf{IF}\ e\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\ \mathsf{FI} \mid \mathsf{WHILE}\ e\ \mathsf{DO}\ c\ \mathsf{OD} \\ \quad \mid \mathsf{TRY}\ c_1\ \mathsf{CATCH}\ c_2\ \mathsf{END} \mid \mathsf{THROW} \mid \mathsf{CALL}\ f \mid \mathsf{GUARD}\ F\ P\ c \mid \mathsf{SPEC}\ r \end{array}$$

The statement BASIC m modifies the state by applying function m to it; in the common case where m is a function that updates a variable a to the value b, we use the notation ´a :== b. GUARD F P c asserts property P before executing c, otherwise aborting execution with fault F. SPEC r non-deterministically selects a new state s' based on the current state s such that $(s, s') \in r$ holds; we use such non-determinism to model hardware and uninitialised memory.

Fig 2(a) shows an example of a simple C function `max` parsed into Simpl. Input parameters a and b are set up by the caller and otherwise treated as local variables, while the return value of the function is recorded in the ghost variable ret-int. The function body is surrounded by an exception handler with the empty SKIP body; this pattern is used to model abrupt termination as in `return`, `break` and `continue`. The ghost variable exn-var records the reason for the current exception, so that, for instance, `return` statements inside loops are not handled by the `break` handler surrounding the loop. Finally, the GUARD command rules out particular undefined behaviour in C; in this case asserting that execution does not fall off the end of the (non-`void`) function. Fig 2(b) is similar in structure, but additionally initialises the variable c to a non-deterministically chosen value.

All Simpl programs execute on a particular state type. In our case it always contains a record with local variables and a record with global variables, among them the heap, a partial function mapping addresses in memory to their byte values. We use Schirmer's notation $\Gamma \vdash \langle C, \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Normal}\ t$ to specify that the Simpl program C starting in state Normal s has at least one execution path resulting in Normal t. Other state types include Abrupt s, indicating the program is currently propagating an exception; Fault f, indicating an irrecoverable failure f; or Stuck, indicating stuck execution. The variable Γ maps function names to function bodies, and is used for making function calls in Simpl. We additionally use Schirmer's notation $\Gamma \vdash C \downarrow \mathsf{Normal}\ s$ to specify that all execution paths of the program C starting in state Normal s terminate.

2.3 Monadic Framework

Our goal is to abstract imperative programs encoded in Simpl into a representation that eases reasoning. But which representation is best suited to such reasoning? Any representation we choose must encode the same functionality as Simpl, including programs that read and write global state; contain loops that

Simpl	Monad	Monadic Definition
–	returnE x	$\lambda s.$ ({(Norm x, s)}, False)
SKIP	skipE	$\lambda s.$ ({(Norm (), s)}, False)
BASIC m	modifyE m	$\lambda s.$ ({(Norm (), $m\ s$)}, False)
THROW	throwE ()	$\lambda s.$ ({(Exc (), s)}, False)
IF c THEN L ELSE R FI	condE $c\ L\ R$	$\lambda s.$ if $c\ s$ then $L\ s$ else $R\ s$
GUARD $t\ g\ B$	guardE g	condE g skipE failE
WHILE c DO B OD	whileE $c\ B$ ()	*(see text)*

Fig. 3. A selection of monadic functions with corresponding Simpl commands

potentially do not terminate; raise and catch exceptions; are non-deterministic; and have execution paths that result in irrecoverable failure.

Our chosen representation is a *state monad* with additional support for non-determinism, exceptions and failure (representing irrecoverable program failure). We name this monad the *exception monad* which has type $'s \Rightarrow ((('e + 'a) \times 's)\ set \times bool$ abbreviated as $('s, 'a, 'e)\ monadE$. The monad accepts a single input state $'s$ and returns a tuple. The first half of this tuple contains the results of the execution: a set of pairs containing a return value and state. The result is a set so that functions may return more than one resulting state, modelling non-determinism. Each return value is either a standard value of type $'a$ indicating normal execution, or an exception value of type $'e$. The second half of the tuple is a flag indicating whether any execution of the monad failed. We name the first and second halves of this tuple results and failed respectively. A full description of the motivation for and formalisation of this monad with VCG support are presented in earlier work [3].

Fig 3 lists the monadic commands used in this paper and their Simpl equivalents where applicable. Monadic commands are suffixed with the character E to indicate they operate on the exception monad. Monadic functions may be joined together by the *bind* operator, where $a \gg\!=_{\mathsf{E}} (\lambda x.\ b\ x)$ denotes that a is executed with its return value passed into b, bound to the variable x. We additionally use the notation doE $x \leftarrow\ a$; $b\ x$ odE as alternative syntax for bind. returnE f simply returns the value f, allowing it to be used later as a bound variable.

To represent loops, we define a combinator whileE $c\ B\ i$ with type:

$$('a \Rightarrow 's \Rightarrow bool) \Rightarrow ('a \Rightarrow ('s, 'a, 'e)\ monadE) \Rightarrow 'a \Rightarrow ('s, 'a, 'e)\ monadE$$

The combinator takes a loop condition c, a loop body B, and an initial *loop iterator* value i. While the condition c remains true, the loop body will be executed with the current loop iterator value. The return value from each iteration of the loop body will become the loop iterator value for the next loop iteration, or the return value for the whileE block if the loop condition is false. This allows us to bind variables in one iteration of the loop and use them in either the next iteration of the loop or after the loop completes. Formally, whileE returns the

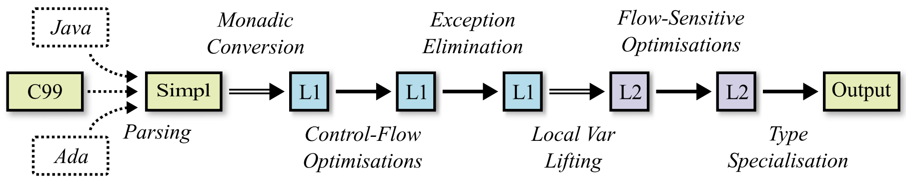


Fig. 4. The process of converting Simpl to an abstracted output program. Dashed arrows represent trusted translations, white arrows represent refinement proofs, while solid arrows represent term rewriting. Each phase beyond parsing is in Isabelle/HOL.

set of all results that can be reached in a finite number of loop iterations. Additionally, whileE fails if any computation within the loop fails or if there exists any non-terminating computation.

3 Abstraction Process

This section describes in detail the transformations that take place from our input Simpl specification to the output specification presented to the user, as well as the proofs of correctness generated at each step. Fig 4 depicts the transformations applied, each of which is described below.

3.1 Conversion to Shallow Embedding

When C code is parsed into Isabelle/HOL, it is converted into the Simpl language with deeply-embedded statements. While such a deep embedding is sufficient for reasoning about program behaviour, in practice it is a frustrating experience: standard Isabelle mechanisms such as term rewriting, which can replace sub-terms of a program with equivalent alternatives, cannot be used, as two semantically equivalent programs are only considered equal if they are *structurally* identical. While tools can be developed to alleviate some of this burden [15], still much of the support provided by Isabelle remains unavailable.

Our first step towards generating an abstraction is thus converting the deeply-embedded Simpl input into a monadic shallow embedding. We name the output of this translation stage *L1*.

The conversion process is conceptually easy: Simpl constructs are simply substituted with their monadic equivalents shown in Fig 3. Our goal, however, is to also generate a proof that the conversion is sound. We achieve this by proving a property $\mathsf{corres}_{\mathsf{L1}}$ stating that the original Simpl program is a refinement of our translated program, defined as follows:

$$
\begin{array}{l}
\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ A\ C \equiv \\
\quad \forall s.\ \neg\ \mathsf{failed}\ (A\ s) \longrightarrow \\
\qquad (\forall t.\ \Gamma \vdash \langle C,\ \mathsf{Normal}\ s\rangle \Rightarrow t \longrightarrow \\
\qquad\quad (\mathsf{case}\ t\ \mathsf{of\ Normal}\ s' \Rightarrow (\mathsf{Norm}\ (),\ s') \in \mathsf{results}\ (A\ s) \\
\qquad\quad\ \mid \mathsf{Abrupt}\ s' \Rightarrow (\mathsf{Exc}\ (),\ s') \in \mathsf{results}\ (A\ s) \mid \text{-} \Rightarrow \mathsf{False})) \wedge \\
\qquad \Gamma \vdash C \downarrow \mathsf{Normal}\ s
\end{array}
$$

$$\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ (\mathsf{modifyE}\ m)\ (\mathsf{BASIC}\ m) \qquad \textsc{L1CorresSkip}$$

$$\frac{\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ L\ L' \qquad \mathsf{corres}_{\mathsf{L1}}\ \Gamma\ R\ R'}{\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ (L \gg\!\!=_{\mathsf{E}} (\lambda y.\ R))\ L';\ R'}\ \textsc{L1CorresSeq}$$

$$\frac{\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ L\ L' \qquad \mathsf{corres}_{\mathsf{L1}}\ \Gamma\ R\ R'}{\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ (\mathsf{condE}\ c\ L\ R)\ (\mathsf{IF}\ c\ \mathsf{THEN}\ L'\ \mathsf{ELSE}\ R'\ \mathsf{FI})}\ \textsc{L1CorresCond}$$

$$\frac{\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ B\ B'}{\mathsf{corres}_{\mathsf{L1}}\ \Gamma\ (\mathsf{whileE}\ (\lambda\text{-}.\ c)\ (\lambda\text{-}.\ B)\ ())\ (\mathsf{WHILE}\ c\ \mathsf{DO}\ B'\ \mathsf{OD})}\ \textsc{L1CorresWhile}$$

Fig. 5. Selection of rules, compositionally proving $\mathsf{corres}_{\mathsf{L1}}$ in the Simpl to L1 translation

The definition reads as follows: Given a Simpl context Γ mapping function names to function bodies, a monadic program A and a Simpl program C, then, assuming that the monadic program A does not fail: (*i*) for each normal execution of the Simpl program there is an equivalent normal execution of the monadic program; (*ii*) similarly, for each execution of the Simpl program that results in an exception, there is an equivalent monadic execution also raising an exception; and, finally (*iii*) every execution of the Simpl program terminates.

The final termination condition may initially seem surprising. Recall, however, that these conditions must only hold if A does not fail, while our definition of whileE ensures that infinite loops will raise the failure flag. Consequently, proving termination of C is reduced to proving non-failure of A.

We prove $\mathsf{corres}_{\mathsf{L1}}$ automatically using a set of syntax-directed rules such as those listed in Fig 5. The final L1 output is a program that has the same structure as the source Simpl program, but is in a more convenient representation.

3.2 Control Flow Peephole Optimisations

The Simpl output generated by the C parser is, by design, as literal a conversion of C as possible. This frequently leads to clutter such as: (*i*) unnecessary skipE statements, generated from stray semicolons (which remain after the preprocessor strips away debugging code); (*ii*) guardE statements that are always true; (*iii*) dead code following throwE or failing guardE statements; or (*iv*) conditional condE statements where the condition is True or False. As the L1 specification is a shallow embedding, we are able to use Isabelle's rewrite engine to apply a series of *peephole optimisations* consisting of 21 rewrite rules, removing significant amounts of unnecessary code from the L1 programs. Table 1 at the end of this paper measures the size reduction in each translation stage.

3.3 Exception Rewriting

Statements in C that cause abrupt termination such as `return`, `continue` or `break` are modelled in Simpl with exceptions, as described in Sec 2.2. While exceptions accurately model the behaviour of abrupt termination, their presence

$$\frac{\textsf{no-throw}\ A}{\textsf{catchE}\ A\ E = A}\ \textsc{CatchNoThrow} \qquad \frac{\textsf{always-throw}\ A}{A \gg\!=_{\textsf{E}} B = A}\ \textsc{SeqAlwaysThrow} \qquad \textsf{catchE}\ (\textsf{throwE}\ a)\ E = E\ a\ \ \textsc{CatchThrow}$$

$$\frac{\textsf{no-throw}\ A}{\textsf{catchE}\ (A \gg\!=_{\textsf{E}} B)\ C = A \gg\!=_{\textsf{E}} (\lambda x.\ \textsf{catchE}\ (B\ x)\ C)}\ \textsc{SeqNoThrow}$$

$$\textsf{catchE}\ (\textsf{condE}\ c\ L\ R)\ E = \textsf{condE}\ c\ (\textsf{catchE}\ L\ E)\ (\textsf{catchE}\ R\ E) \quad \textsc{CatchCond}$$

$$\begin{aligned}&\textsf{catchE}\ (\textsf{condE}\ C\ L\ R \gg\!=_{\textsf{E}} B)\ E\\ &= \textsf{condE}\ C\ (\textsf{catchE}\ (L \gg\!=_{\textsf{E}} B)\ E)\ (\textsf{catchE}\ (R \gg\!=_{\textsf{E}} B)\ E)\end{aligned} \quad \textsc{CatchCondSeq}$$

Fig. 6. Rewrite rules to reduce exceptions in control flow

complicates reasoning about the final program: each block of code now has *two* exit paths that must be considered.

Fortunately, most function bodies can be rewritten to avoid the use of exceptions. Fig 6 shows the set of rewrite rules we use to reduce exceptional control flow. CatchNoThrow eliminates exception handlers surrounding code that never raises exceptions (denoted by no-throw). Analogously, SeqAlwaysThrow removes code trailing a block that *always* raises an exception (denoted by always-throw). The no-throw and always-throw side-conditions are proved automatically using a syntax-directed set of rules.

Not all rules in this set can be applied blindly. In particular, the rules CatchCond and CatchCondSeq duplicate blocks of code, which may trigger exponential growth in pathological cases. For CatchCond, which duplicates the exception handler, knowledge of our problem domain saves us: inputs originating from C only have trivial exception handlers generated by the parser, and hence duplicating them is of no concern.

The rule CatchCondSeq, however, also duplicates its tail B, which may be arbitrarily large. We carry out the following steps to avoid duplication: (*i*) if neither branch of the condition throws an exception, then SeqNoThrow is applied; (*ii*) if both branches throw an exception, then SeqAlwaysThrow is applied; (*iii*) if one branch always throws an exception, then the rule CatchCondSeq is applied followed by SeqAlwaysThrow on that branch, resulting in only a single instance of B in the output; finally (*iv*) if the body B is trivial, such as a simple returnE or throwE statement, we apply CatchCondSeq and duplicate B under the assumption the rewritten specification will still be simpler than the original. Otherwise, we leave the specification unchanged, and let the user reason about the exception rather than a larger output specification.

Using these rules, all exceptions can be eliminated other than those in nested condition blocks described above, or those caused by `break` or `return` statements inside loop bodies. Applying the transformation to the seL4 microkernel, 96% of functions could be rewritten to eliminate exceptional control flow. Of the remaining 4%, 10 could not be rewritten due to nested condition blocks, 13

```
doE
  modifyE (λs. s(| a-' := 3 |));
  condE (λs. 5 ≤ a-' s)
      (modifyE (λs. s(| b-' := 5 |)))
      (modifyE (λs. s(| c-' := 4 |)));
  modifyE (λs. s(| ret-int-' := a-' s |))
odE
```

(a) Locals in state

```
doE
  a ← returnE 3;
  (b, c) ← condE (λs. 5 ≤ a)
      (returnE (5, c))
      (returnE (b, 4));
  returnE a
odE
```

(b) Local lifted form

Fig. 7. Two program listings. The first stores locals in the state, while the second uses bound variables. The shaded region does not affect the final return value; this is clearly apparent in the second representation.

because of either `return` or `break` statements inside a loop, and one function for both reasons independently.

3.4 Local Variable Lifting

Both the Simpl embedding of our original input programs and our L1 translation represent local variables as part of the state: each time a local is read it is extracted from the state, and each time a local is written the state is modified. While this representation is easy to generate, it complicates reasoning about variable usage. An example of this is shown in Fig 7(a): the variable a is set to the value 3 at the top of the function and later returned by the function. However, to prove that the function returns the value 3, the user must first prove that the shaded part of the program preserves a's value.

An alternative approach to representing locals is using the bound variables feature provided by our monadic framework that we have so far ignored. To achieve this, we remove locals from the state type and instead model them as bound Isabelle/HOL variables. We name this representation *lifted local form* and the output of this translation *L2*. The representation is analogous to *static single-assignment* (SSA) form used by many compilers as an intermediate representation [9], where each variable is assigned precisely once.

Fig 7(b) shows the same program in lifted local form. The function returns the variable a, which is bound to the value 3 in the first line of the function. As variables cannot change once bound, the user can trivially determine that the function returns 3 without inspecting the shaded area.

Two complications arise in representing programs in local lifted form. The first is that variables bound inside the bodies of condE and catchE blocks are not available to statements after the block. To overcome this, we modify the bodies of such blocks to return a tuple of all variables modified in the bodies and subsequently referenced, as demonstrated in Fig 7(b); statements following the block can then use the names returned in this tuple. The second, similar complication arises from loops, where locals bound in one iteration not only need to be accessible after the loop, but also accessible by statements in the *next* iteration. We solve this by passing all required locals between successive iterations of the loop as well as the result of the loop in the iterator of the whileE combinator. The gcd function in Fig 1 shows an example. In both cases, the tool must perform program analysis to determine which variables are modified. The emitted proofs imply correctness of this analysis as we shall see below.

For the soundness proof of the translation from L1 to L2 we use a refinement property $\mathsf{corres}_{\mathsf{L2}}$ defined as follows:

$$
\begin{array}{l}
\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ A\ C \equiv \\
\quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ (st\ s)) \longrightarrow \\
\qquad (\forall (r,\ t){\in}\mathsf{results}\ (C\ s). \\
\qquad\quad \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Exc}\ ()\ \Rightarrow (\mathsf{Exc}\ (ex\ t),\ st\ t) \in \mathsf{results}\ (A\ (st\ s)) \\
\qquad\quad \mid \mathsf{Norm}\ ()\ \Rightarrow (\mathsf{Norm}\ (rx\ t),\ st\ t) \in \mathsf{results}\ (A\ (st\ s))) \wedge \\
\qquad \neg\ \mathsf{failed}\ (C\ s)
\end{array}
$$

The predicate has several parameters: st is a state translation function, converting the L1 state type to the L2 state type by stripping away local variable data; P is a precondition used to ensure that input bound variables in the L2 program match their L1 values; and A and C are the abstract L2 and concrete L1 programs respectively. The values rx and ex are a *return extraction function* and an *exception extraction function* respectively; they are required because the L2 monads return or throw variables, while the corresponding L1 monads store these values in their state. The return extraction function rx extracts a value out of the L1 state to compare with the return value of the L2 monad, while ex is used to compare an exception's payload with the corresponding L1 state.

The $\mathsf{corres}_{\mathsf{L2}}$ definition can be read as: for all states matching the precondition P, assuming that A executing from state $st\ s$ does not fail, then the following holds: (i) for each normal execution of C there is an equivalent execution of A whose return value will match the value extracted using rx from C's state; (ii) similarly, every exceptional execution of the C will have an equivalent execution of A with an exception value that matches the value extracted using ex from C's state; and, finally (iii) the execution of C will not fail.

The first two conditions ensure that executions in L2 match those of L1 with locals bound accordingly. The last condition allows us to later reduce non-failure of L1 programs to non-failure of L2 programs.

As a concrete example, Fig 9 shows our example max function after local variable lifting has taken place. The generated $\mathsf{corres}_{\mathsf{L2}}$ predicate for max is:

$$\mathsf{corres}_{\mathsf{L2}}\ \mathsf{globals}\ \mathsf{ret\text{-}int\text{-}}'\ (\lambda s.\ ())\ (\lambda s.\ \mathsf{a\text{-}}'\ s = a \wedge \mathsf{b\text{-}}'\ s = b)\ (\mathsf{max}_{\mathsf{L2}}\ a\ b)\ \mathsf{max}_{\mathsf{L1}}$$

In this example the state translation function globals strips away local variables from the L1 state; the return extraction function rx ensures the value returned by $\mathsf{max}_{\mathsf{L2}}$ matches the variable $\mathsf{ret\text{-}int\text{-}}'$ of $\mathsf{max}_{\mathsf{L1}}$, while the exception extraction function ex is unused and simply returns unit, as no exceptions are thrown by the max function. The remainder of the predicate states that, assuming the inputs a and b to our $\mathsf{max}_{\mathsf{L2}}$ function match those of the L1 state, then the return value of our $\mathsf{max}_{\mathsf{L2}}$ function will match the L1 state variable ret-int after executing $\mathsf{max}_{\mathsf{L1}}$.

$$\frac{\forall s.\ P\ s \longrightarrow st\ s = st\ (M'\ s) \qquad \forall s.\ P\ s \longrightarrow rx\ (M'\ s) = v}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ (\mathsf{returnE}\ v)\ (\mathsf{modifyE}\ M')}\ \textsc{L2CorresReturn}$$

$$\frac{\forall s.\ P\ s \longrightarrow M\ (st\ s) = st\ (M'\ s)}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ (\mathsf{modifyE}\ M)\ (\mathsf{modifyE}\ M')}\ \textsc{L2CorresModify}$$

$$\frac{\begin{array}{c}\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ Q_A\ A\ A' \qquad \forall x.\ \mathsf{corres}_{\mathsf{L2}}\ st\ rx'\ ex\ (Q_B\ x)\ (B\ x)\ B' \\ \{\!|P|\!\}\ A'\ \{\!|\lambda\text{-}\ s.\ Q_B\ (rx\ s)\ s|\!\},\ \{\!|\lambda\text{-}\ \text{-}.\ \mathsf{True}|\!\} \qquad \forall s.\ P\ s \longrightarrow Q_A\ s\end{array}}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx'\ ex\ P\ (A \gg\!\!=_{\mathsf{E}} B)\ (A' \gg\!\!=_{\mathsf{E}} (\lambda x.\ B'))}$$
$$\textsc{L2CorresSeq}$$

$$\frac{\begin{array}{c}\forall x.\ \mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ (Q'\ x)\ (A\ x)\ B \\ \{\!|\lambda s.\ Q\ (rx\ s)\ s|\!\}\ B\ \{\!|\lambda\text{-}\ s.\ Q\ (rx\ s)\ s|\!\},\ \{\!|\lambda\text{-}\ \text{-}.\ \mathsf{True}|\!\} \\ \forall s.\ Q\ (rx\ s)\ s \longrightarrow c'\ s = c\ (rx\ s)\ (st\ s) \\ \forall s\ x.\ Q\ x\ s \longrightarrow Q'\ x\ s \qquad \forall s.\ Q\ x\ s \longrightarrow rx\ s = x \qquad \forall s.\ P\ x\ s \longrightarrow Q\ x\ s\end{array}}{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ (P\ x)\ (\mathsf{whileE}\ c\ A\ x)\ (\mathsf{whileE}\ (\lambda\text{-}.\ c')\ (\lambda\text{-}.\ B)\ ())}$$
$$\textsc{L2CorresWhile}$$

Fig. 8. Selected rules used in the $\mathsf{corres}_{\mathsf{L2}}$ proofs

We prove the predicate $\mathsf{corres}_{\mathsf{L2}}$ compositionally using a syntax-directed approach similar to our rule set for $\mathsf{corres}_{\mathsf{L1}}$. Fig 8 shows a sample of the rules used to carry out the proofs. We use the Hoare-style syntax $\{\!|P|\!\}\ C\ \{\!|Q|\!\}, \{\!|E|\!\}$ to state that program C starting in a state satisfying P ensures Q in the event of normal termination or E in the event of an exception.

The rule L2CorresReturn shows that the L1 statement $\mathsf{modifyE}\ M'$ refines the L2 statement $\mathsf{returnE}\ v$ if the state-update function M' only modifies locals, and the L2 return value v corresponds to the local updated in L1, extracted using rx. The rule L2CorresModify is similar, but is used when an L1 $\mathsf{modifyE}$ statement updates non-local state. Automating such proofs requires: (i) parsing the term M'; (ii) determining if it has a local or non-local effect; (iii) emitting the corresponding abstract statement; and finally (iv) generating the corresponding proof term. If an L2 term is correctly generated then the side-conditions of the rule are discharged automatically by Isabelle's simplifier.

The composition rules are more involved. For instance, L2CorresSeq states the L1 program fragment $A' \gg\!\!=_{\mathsf{E}} (\lambda\text{-}.\ B')$ refines the L2 program fragment $A \gg\!\!=_{\mathsf{E}} B$. For the rule to apply, A' must refine A under the precondition Q_A, and B' must refine B under precondition Q_B. This latter precondition has an additional parameter x representing the return value from A. We must prove that executing A' from a state satisfying P leads to a state s where the second precondition $Q_B\ (rx\ s)$ is satisfied. This second parameter to Q_B is what ensures that the locals stored in the L1 state match the bound variables used in L2.

To automatically prove an application of the L2CorresSeq rule, we must calculate a suitable precondition P that both implies the first precondition Q_A and will lead to Q_B being satisfied. We generate such a P stating that all bound

```
condE (λs. a <_s b)
   (doE
      ret ← returnE b;
      exn-var ← returnE Return;
      returnE ret
    odE)
   (doE
      ret ← returnE a;
      exn-var ← returnE Return;
      returnE ret
    odE)
```

Fig. 9. The `max` function after local variable lifting

```
condE (λs. a <_s b)
   (returnE b)
   (returnE a)
```

Fig. 10. The `max` function after flow-sensitive optimisations

```
returnE ( if a <_s b then b else a )
```

Fig. 11. The `max` function after type-strengthening

variables required by A match their L1 state; and all bound variables required by B and not modified by A match their L1 state. Using this P, we can discharge the Hoare-style side-condition by showing that A' preserves all variables required by B which it does not otherwise pass in by bound variables; these proofs are again automated using a syntax-directed rule-set.

3.5 Flow-Sensitive Simplifications

A significant benefit of lifted local form is that it allows us to easily determine how local variables are used, and carry out simplifications based on this. Such simplifications include: (*i*) removing code that writes to locals that are never subsequently read from; (*ii*) using assumptions from guardE, condE and whileE statements to simplify later expressions; and (*iii*) collapsing variables that are only used once into the locations where they are used. By allowing constant valued expressions to be folded into the location they are used, we are also able to discharge many more guardE statements not previously provable and determine that some condE conditions always have the same value.

Fig 10 shows the `max` function after flow-sensitive optimisations: the redundant exn-var variable is detected, and the two returnE terms in each branch of the condE are collapsed into a single statement, resulting in a much simpler program.

3.6 Type Specialisation

So far, all of our generated programs have been written using our exception monad. Sec 2.3 outlined some of the motivations for using this monad, including our aim to represent C that supports reading and writing from global state; abrupt termination; non-determinism; or code that may fail.

For the majority of the code we translate, many of these features are not required. For example, Sec 3.3 describes how the majority of exception usage can be eliminated from specifications; the use of non-determinism is mostly limited to setting up uninitialised variables, many of which are eliminated using the simplifications in Sec 3.5; further, many functions do not modify the state of the system at all, either only reading the global state or having results that depend entirely on their input parameters. In these cases, the exception monad is far more expressive than required. Less expressive monads would constrain program behaviour by type and give the user free theorems by notation alone.

We therefore specialise the type of individual functions to contain only the features they require. The types we use are as follows, in decreasing strength:

Pure functional: These are standard Isabelle functions, where the function returns a deterministic output depending only on its input parameters. Our example `max` function falls into this category.

Option monad: The C standard is littered with restrictions that result in guardE statements that cannot be automatically discharged. Unfortunately, such a guardE statement will prevent a function from being translated into a pure Isabelle function, as we must consider failed executions. We can, however, use the *option monad*, where every computation either results in a single value a (represented as $\mathsf{Some}\ a$), or failure (represented as None). Any intermediate failure results in failure of the entire computation.

Functions that may potentially fail, but are deterministic, have simple control flow, and only read from global state can be transformed into the option monad. Callers of such functions will translate a result of None into failure.

State monad: Functions which need to modify global state or use non-determinism but do not use exceptional control flow are translated into a state monad without exceptions.

Type specialisation takes place using a series of rewrite rules that attempt to strengthen individual parts of the program, and then combine partial results to strengthen larger parts of the program. For instance, the rewrite rules we use to strengthen the exception monad to a pure Isabelle function are as follows:

$$\begin{aligned}
\mathsf{skipE} &= \mathsf{returnE}\ ()\\
\mathsf{condE}\ (\lambda\text{-}.\ c)\ (\mathsf{returnE}\ A)\ (\mathsf{returnE}\ B) &= \mathsf{returnE}\ (\mathsf{if}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B)\\
\mathsf{returnE}\ A \gg\!=_{\mathsf{E}} (\lambda x.\ \mathsf{returnE}\ (B\ x)) &= \mathsf{returnE}\ (\mathsf{let}\ x = A\ \mathsf{in}\ B\ x)
\end{aligned}$$

Fig 11 shows the result of applying these rules to our example `max` function.

To determine which type we can strengthen each function to, we attempt to apply each set of strengthening rules in order from strongest type to weakest type. If a particular function can be completely rewritten, the transformation was successful and a new definition for the function is emitted. Otherwise, we continue to try alternative, more expressive, representations. If all translations fail, we simply continue to use the exception monad.

Table 2 shows statistics of type strengthening used on the seL4 microkernel source code, with almost 96% of functions being strengthened into another type. The remaining 4% of functions correspond to the functions unable to be rewritten to avoid using exceptions in Sec 3.3.

Table 1. Average function term size after each translation phase of the seL4 source

Specification	Avg. Size
Simpl	356.7
Shallow Embedding	362.5
Control-Flow Peephole	286.2
Exception Rewriting	281.1
Lifted Local Vars	249.2
Flow-Sensitive Opts.	173.6
Type Strengthening	173.5
Polish	168.9

Table 2. Number of functions in the seL4 microkernel translated into each type

Type	Count
Pure function	151
Option monad	51
State monad	309
Exception monad	24
Total	535

3.7 Final Theorem

Along with the abstracted program specification, the tool emits a proof of correctness. In particular, the final refinement theorem between the input Simpl and output monadic program is as follows:

$$\begin{array}{l}
\mathsf{corres}_{\mathsf{F}}\ st\ \Gamma\ rx\ P\ A\ C \equiv \\
\quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ (st\ s)) \longrightarrow \\
\qquad (\forall t.\ \Gamma \vdash \langle C, \mathsf{Normal}\ s\rangle \Rightarrow t \longrightarrow \\
\qquad\quad (\exists s'.\ t = \mathsf{Normal}\ s' \wedge \\
\qquad\qquad (\mathsf{Norm}\ (rx\ s'), st\ s') \in \mathsf{results}\ (A\ (st\ s)))) \wedge \\
\qquad \Gamma \vdash C \downarrow \mathsf{Normal}\ s
\end{array}$$

It composes the two previous theorems, and is proved with the following rule:

$$\frac{\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ (\lambda\text{-}.\ ())\ P\ A\ B \qquad \mathsf{corres}_{\mathsf{L1}}\ \Gamma\ B\ C \qquad \mathsf{no\text{-}throw}\ A}{\mathsf{corres}_{\mathsf{F}}\ st\ \Gamma\ rx\ P\ A\ C}$$

Our final $\mathsf{corres}_{\mathsf{F}}$ definition, while differing from the definition given in previous work on C abstraction in L4.verified [15], is strong enough to prove it.

In summary, we have shown the following transformations: (*i*) from deep into shallow embedding, which enables us to use rewriting; (*ii*) simple control flow peephole rewrites, exploiting the shallow embedding; (*iii*) exception rewriting, which further simplifies control flow; (*iv*) local variable lifting, which allows us to make use of Isabelle's built-in bound variables, substitution, and unification; (*v*) flow-sensitive rewrites, enabled by explicit bound variables; (*vi*) and, type specialisation, giving the user convenient notation and implicit free theorems.

The final additional step is a polishing phase which rewrites internal terms into a more human-friendly form. Table 1 quantifies the effect of each transformation by showing the average term size after each phase for the translation of seL4.

4 Related Work

The motivation for this work is the paper *Mind the Gap: A verification framework for low-level C* by Winwood et al. [15] in the context of the seL4 microkernel verification. They showed that formal, interactive verification of low-level C code at scale is possible, but noted that automation could be improved. Our final refinement theorem implies their ccorres statement. Some of the automation ideas are present in early forms in this previous work, such as lifting a single variable from the state into a bound variable in the monad. In addition to our other transformations, we generalise this approach to completely automatically lift all variables of all functions in a program. On the technical side, our rule sets and refinement statements are tuned for full automation. The idea is to remove all unnecessary manual work in low-level C verification and enable the human to concentrate on the interesting reasoning instead.

Two further projects have treated large low-level code bases interactively. The Verisoft project [1] reasoned directly about Simpl using a VCG that translates Hoare triples about Simpl code into proof obligations. While the VCG provides some automation, it performs less abstraction. Consequently, the verification overhead in Verisoft was similarly high as in L4.verified.

The Verisoft-XT project applied the VCC tool [4]. VCC does not attempt automated abstraction of this form either, but instead uses a powerful SMT solver as backend reasoner to increase productivity. The increased automation comes at the cost of reduced expressiveness in annotations and explicit ghost state to guide the reasoner. While our focus is on interactive reasoning, we believe the approach is complementary: our tool could be used to generate a higher-level, less detailed model, and automated reasoners could then be used on top.

The FramaC framework [5] with the Jessie plug-in [8] also supports deductive verification of C. Annotated C code is translated into the functional language Why on which verification then proceeds. The translation touches on some transformations that are close to ours. The main difference is that these transformations need to be trusted whereas our work produces proofs. The necessarily trusted translation step from C into a formal logic is much smaller in our work.

5 Conclusions

We have presented a tool that automatically abstracts low-level C semantics into higher-level specifications with automatic proofs of correctness for each of the transformation steps. The tool consists of 3 300 lines of ML code and 5 000 lines of Isabelle proof script, on top of existing libraries for monads, Simpl and parsing.

While our main case study is the seL4 microkernel, because it provides a convenient known target for comparison, the tool is not specific to this kernel. We have also applied it to Tuch's memory allocator case study [14], and other projects such as the scheduler of a small commercial real-time system. We believe that the general idea can be applied to languages other than C, and that the tool may even be directly applicable to these as long as a front-end to Simpl exists.

The tool accepts anything the C parser front-end accepts, but presently does not translate recursive functions. While not a problem for embedded code, this is one of the obvious next steps for future work. The second direction for future work is to provide further translation steps, for instance exploiting Tuch's interactive framework [13] to automatically generate a more abstract heap format for type safe fragments of the program.

Our experience indicates a significant improvement in clarity and ease of reasoning for the output of the tool. Our long term goal is to completely automate the low-level C verification phase in Winwood et al. [15] for projects like L4.verified.

Acknowledgements. We are grateful to Matthias Daum, Daniel Matichuk, Thomas Sewell and the anonymous reviewers for their feedback on drafts of this paper.

References

1. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A., Tsyban, A.: Balancing the load — leveraging a semantics stack for systems verification. JAR: Special Issue Operat. Syst. 42(2-4), 389–454 (2009)
2. Alkassar, E., Paul, W.J., Starostin, A., Tsyban, A.: Pervasive Verification of an OS Microkernel. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Heidelberg (2010)
3. 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)
4. 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)
5. The Frama-C platform (2008), http://www.frama-c.cea.fr/
6. ISO/IEC. Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14 (May 2005)
7. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSP, pp. 207–220. ACM (2009)
8. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. PhD thesis, Université Paris-Sud, Paris, France (January 2009)
9. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers (1997)
10. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
11. Norrish, M.: C-to-Isabelle parser, version 0.7.2 (January 2012), http://ertos.nicta.com.au/software/c-parser/

12. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
13. Tuch, H.: Formal Memory Models for Verifying C Systems Code. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia (August 2008)
14. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) 34th POPL, pp. 97–108. ACM (2007)
15. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the Gap: A Verification Framework for Low-Level C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 500–515. Springer, Heidelberg (2009)

Abstract Interpretation of Annotated Commands

Tobias Nipkow[⋆]

Fakultät für Informatik, Technische Universität München

Abstract. This paper formalizes a generic abstract interpreter for a while-language, including widening and narrowing. The collecting semantics and the abstract interpreter operate on annotated commands: the program is represented as a syntax tree with the semantic information directly embedded, without auxiliary labels. The aim of the paper is simplicity of the formalization, not efficiency or precision. This is motivated by the inclusion of the material in a theorem prover based course on semantics.

1 Introduction

The purpose of this work is to formalize the basics of abstract interpretation in a theorem prover in as simple a manner as possible. The background is a course on semantics [10] that is completely based on Isabelle/HOL [11]. The first 4 weeks of the course are dedicated to the theorem prover; the rest of the course focuses on the semantics of a simple while-language and on its applications (e.g. compiler correctness). In particular, the last 4 weeks are dedicated to abstract interpretation. Hence the need to concentrate on the essence and simplify the technicalities. A second desideratum was to stick with the unifying representation of programs as abstract syntax trees employed throughout the course. Finally we wanted to visualize the stepwise computation of the semantics and the abstract interpreter as directly as possible. As a result we chose syntax trees annotated with (concrete or abstract) semantic information and a Jacobi-like iteration strategy. That is, displaying the annotated program after each iteration step animates the stepwise approximation of the result. This paper presents the formalization of a collecting semantics, a derived small-step operational semantics, and a stepwise development of a series of abstract interpreters, up to and including widening and narrowing. Just like previous formalizations, we only consider concretization, not abstraction, and verify only correctness, not optimality of the interpreter. Due to space limitations, this is not a tutorial paper and readers are assumed to be familiar with abstract interpretation [4,5,8].

Abstract interpretation is a vast research area, but only a few formalizations have been published, primarily the impressive work by Pichardie [12,13,3], who employs Coq's expressive type and module system to great effect. The key differences to our approach are that Pichardie labels the nodes of the syntax tree

⋆ Research supported by NICTA and by DFG GK PUMA.

whereas we annotate the tree directly with information, his whole approach is denotational (i.e. nested iterations) whereas ours is based on one global iteration, the termination proofs for widening and narrowing are very different, and overall his model is more refined and ours is simpler, which reflects the different aims. Bertot [1] presents an approach that is also based on annotating the program directly but is otherwise very different from ours: Bertot's reference point is a Hoare logic, not a collecting semantics. There have also been a number of specific applications of abstract interpretation, eg [9,2], but without a formalization of the generic theory.

2 Notation

The logic HOL of the Isabelle proof assistant conforms largely to everyday mathematical notation. This section summarizes non-standard notation.

The function space is denoted by $\Rightarrow$. Type variables are denoted by $'a$, $'b$, etc. The notation $t :: \tau$ means that term t has type τ. Type constructors follow postfix syntax, eg $'a\ set$ is the type of sets of elements of type $'a$. Lists over type $'a$, type $'a\ list$, come with the empty list $[]$, the infix constructor $\cdot$, and enumeration syntax $[x_1, \ldots, x_n]$. The **datatype** $'a\ option = None \mid Some\ 'a$ is predefined. The notation $[\![\ A_1, \ldots, A_n\ ]\!] \Longrightarrow B$ is an implication with the premises A_i and the conclusion B.

3 Annotated Commands

There are arithmetic and boolean expressions, where *vname* = *string*:

datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$
datatype $bexp = Bc\ bool \mid Not\ bexp \mid And\ bexp\ bexp \mid Less\ aexp\ aexp$

Their evaluation is defined as usual: $aval :: aexp \Rightarrow state \Rightarrow int$ and $bval :: bexp \Rightarrow state \Rightarrow bool$, where $state = vname \Rightarrow int$. There are commands (type *com*) and *annotated commands*, with the customary concrete syntax; annotations of type $'a$ are enclosed in braces:

```
datatype 'a acom =
           SKIP { 'a }
         | string ::= aexp { 'a }
         | 'a acom ; 'a acom
         | IF bexp THEN 'a acom ELSE 'a acom { 'a }
         | { 'a } WHILE bexp DO 'a acom { 'a }
```

Type *com* is not shown as it is identical to *acom*, but without the annotations.

Annotations positioned at the end of a command refer to the very end of that command, not to some subcommand (eg the *ELSE* branch or the *WHILE* body). The annotation in front of *WHILE* is meant to hold the invariant.

There are many alternatives as to the placement and number of annotations. Our choice fits our formalization of semantics and abstract interpretation, but other choices are possible.

There are a number of auxiliary functions: $\mathit{post} :: {}'a\ \mathit{acom} \Rightarrow {}'a$ extracts the post-annotation of a command ($\mathit{post}\ (c_1;\ c_2) = \mathit{post}\ c_2$), $\mathit{strip} :: {}'a\ \mathit{acom} \Rightarrow \mathit{com}$ removes all annotations, and $\mathit{anno} :: {}'a \Rightarrow \mathit{com} \Rightarrow {}'a\ \mathit{acom}$ annotates a command with the same annotation everywhere.

We say that c_1 and c_2 are *strip-equal* if $\mathit{strip}\ c_1 = \mathit{strip}\ c_2$.

4 Collecting Semantics

The purpose of the collecting semantics is to collect the set of all reachable states at some program point as an annotation. Both the collecting semantics and later the abstract interpreter are defined by iterated simultaneous "micro-step" execution of all atomic commands, similar to the Jacobi method for linear equations. This is very different from a denotational approach where whole subcommands are executed in one go. We define a function $\mathit{step} :: \mathit{state\ set} \Rightarrow \mathit{state\ set\ acom} \Rightarrow \mathit{state\ set\ acom}$ that pushes a set of initial states one step into an annotated command c and propagates the *state set* annotations inside c one step further:

$$
\begin{array}{ll}
\mathit{step}\ S\ (\mathit{SKIP}\ \{P\}) = \mathit{SKIP}\ \{S\} & \\
\mathit{step}\ S\ (x ::= e\ \{P\}) = x ::= e\ \{\{s' \mid \exists s{\in}S.\ s' = s(x := \mathit{aval}\ e\ s)\}\} & \\
\mathit{step}\ S\ (c_1;\ c_2) = \mathit{step}\ S\ c_1;\ \mathit{step}\ (\mathit{post}\ c_1)\ c_2 & \\
\mathit{step}\ S\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{P\}) = & \mathit{IF}\ b\ \mathit{THEN}\ \mathit{step}\ \{s \in S \mid \mathit{bval}\ b\ s\}\ c_1 \\
& \mathit{ELSE}\ \mathit{step}\ \{s \in S \mid \neg\ \mathit{bval}\ b\ s\}\ c_2 \\
& \{\mathit{post}\ c_1 \cup \mathit{post}\ c_2\} \\
\mathit{step}\ S\ (\{\mathit{Inv}\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}) = & \{S \cup \mathit{post}\ c\} \\
& \mathit{WHILE}\ b\ \mathit{DO}\ \mathit{step}\ \{s \in \mathit{Inv} \mid \mathit{bval}\ b\ s\}\ c \\
& \{\{s \in \mathit{Inv} \mid \neg\ \mathit{bval}\ b\ s\}\}
\end{array}
$$

Annotations for *IF* and *WHILE* are (in principle) redundant, but the invariant is conceptually important and the post-annotations allow a uniform definition of *post* for arbitrary annotations.

The beauty of annotated commands is the ability to visualize the semantics by evaluating *step*. This is possible thanks to Isabelle's evaluation mechanism, which can handle finite sets. Here is a small (contrived) example, further examples follow. Given the command *cs-ex* =

$$
\begin{array}{l}
''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 1)\ \{\{\lambda x.\ 5,\ \lambda x.\ 6,\ \lambda x.\ 7\}\}; \\
''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 2)\ \{\emptyset\}
\end{array}
$$

evaluation of $\mathit{show\text{-}acom}\ [''x'']\ (\mathit{step}\ \{\lambda x.\ -1,\ \lambda x.\ 1\}\ \mathit{cs\text{-}ex})$ yields

$$
\begin{array}{l}
''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 1)\ \{\{[(''x'',\ 0)],\ [(''x'',\ 2)]\}\}; \\
''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 2)\ \{\{[(''x'',\ 7)],\ [(''x'',\ 8)],\ [(''x'',\ 9)]\}\}
\end{array}
$$

In the input, states are functions, but in the output, the pretty-printing function *show-acom* converts states into variable-value pairs, for a given list of variables.

In order to find least fixed-points of *step*, we extend orderings $\leq$ on type *'a* to *'a acom*:

$$\begin{array}{ll}
\mathit{SKIP}\ \{S\} \leq \mathit{SKIP}\ \{S'\} & \longleftrightarrow\ S \leq S' \\
x ::= e\ \{S\} \leq x' ::= e'\ \{S'\} & \longleftrightarrow\ x = x' \wedge e = e' \wedge S \leq S' \\
c_1;\ c_2 \leq d_1;\ d_2 & \longleftrightarrow\ c_1 \leq d_1 \wedge c_2 \leq d_2 \\
\multicolumn{2}{l}{\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{S\} \leq \mathit{IF}\ b'\ \mathit{THEN}\ d_1\ \mathit{ELSE}\ d_2\ \{S'\}} \\
\multicolumn{2}{l}{\longleftrightarrow\ b = b' \wedge c_1 \leq d_1 \wedge c_2 \leq d_2 \wedge S \leq S'} \\
\multicolumn{2}{l}{\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\} \leq \{I'\}\ \mathit{WHILE}\ b'\ \mathit{DO}\ c'\ \{P'\}} \\
\multicolumn{2}{l}{\longleftrightarrow\ b = b' \wedge c \leq c' \wedge I \leq I' \wedge P \leq P'}
\end{array}$$

In all other cases $c \leq c'$ is defined to be *False*. We can now compare commands annotated with state sets. The underlying ordering on the state sets is $\subseteq$. A simple inductive proof shows monotonicity of *step*:

Lemma. *If* $c_1 \leq c_2$ *and* $S_1 \subseteq S_2$ *then* $\mathit{step}\ S_1\ c_1 \leq \mathit{step}\ S_2\ c_2$.

To show that *step* has a least fixed point we turn *acom* into a complete lattice.

4.1 Indexed Complete Lattices

Only subsets of *acom* form a complete lattice, namely $\{c' \mid \mathit{strip}\ c' = c\}$ for any c. Hence we define a little theory of *indexed* complete lattices parameterized by

$$L :: {'i} \Rightarrow {'a}\ \mathit{set} \quad \text{and} \quad \mathit{Glb} :: {'i} \Rightarrow {'a}\ \mathit{set} \Rightarrow {'a}$$

where *'i* is the index type and $L\ i$ the carrier set. We assume that *Glb* is the greatest lower bound and that $L\ i$ is closed under *Glb*:

$$\begin{array}{l}
[\![A \subseteq L\ i;\ a \in A]\!] \Longrightarrow \mathit{Glb}\ i\ A \leq a \\
[\![b \in L\ i;\ \forall a{\in}A.\ b \leq a]\!] \Longrightarrow b \leq \mathit{Glb}\ i\ A \\
A \subseteq L\ i \Longrightarrow \mathit{Glb}\ i\ A \in L\ i
\end{array}$$

In this context we can prove that $\mathit{lfp}\ f\ i = \mathit{Glb}\ i\ \{a \in L\ i \mid f\ a \leq a\}$ is indeed the least fixed and post-fixed point. Note that we define *post-fixed point* to mean $f\ x \leq x$, which is customary in the abstract interpretation literature, although usually this is called a pre-fixed point.

4.2 Application to Collecting Semantics

The Glb of a set of annotated commands is taken pointwise, assuming the commands are all *strip*-equal. More generally, any function on annotation sets can be lifted to sets of annotated commands in this pointwise manner (where f ‘ M is the image of a function over a set):

$$\begin{array}{l}
\mathit{lift} :: ({'a}\ \mathit{set} \Rightarrow {'a}) \Rightarrow \mathit{com} \Rightarrow {'a}\ \mathit{acom}\ \mathit{set} \Rightarrow {'a}\ \mathit{acom} \\
\mathit{lift}\ F\ \mathit{SKIP}\ M = \mathit{SKIP}\ \{F\ (\mathit{post}\ \text{‘}\ M)\} \\
\mathit{lift}\ F\ (x ::= a)\ M = x ::= a\ \{F\ (\mathit{post}\ \text{‘}\ M)\}
\end{array}$$

$$\begin{aligned}
&\mathit{lift}\ F\ (c_1;\ c_2)\ M = \mathit{lift}\ F\ c_1\ (\mathit{sub}_1\ `\ M);\ \mathit{lift}\ F\ c_2\ (\mathit{sub}_2\ `\ M)\\
&\mathit{lift}\ F\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2)\ M = \mathit{IF}\ b\ \mathit{THEN}\ \mathit{lift}\ F\ c_1\ (\mathit{sub}_1\ `\ M)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{ELSE}\ \mathit{lift}\ F\ c_2\ (\mathit{sub}_2\ `\ M)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \{F\ (\mathit{post}\ `\ M)\}\\
&\mathit{lift}\ F\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ M = \{F\ (\mathit{invar}\ `\ M)\}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{WHILE}\ b\ \mathit{DO}\ \mathit{lift}\ F\ c\ (\mathit{sub}_1\ `\ M)\\
&\qquad\qquad\qquad\qquad\qquad \{F\ (\mathit{post}\ `\ M)\}
\end{aligned}$$

Subcommands and the invariant are accessed by auxiliary functions:

$$\begin{aligned}
&\mathit{sub}_1\ (c_1;\ c_2) = c_1\\
&\mathit{sub}_1\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{S\}) = c_1\\
&\mathit{sub}_1\ (\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}) = c\\
&\mathit{sub}_2\ (c_1;\ c_2) = c_2\\
&\mathit{sub}_2\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{S\}) = c_2\\
&\mathit{invar}\ (\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}) = I
\end{aligned}$$

Lemma. Type *'a set acom* is a complete lattice indexed by *com* where $L\ c = \{c' \mid \mathit{strip}\ c' = c\}$ and $\mathit{Glb} = \mathit{lift}\ \bigcap$.

Of course this works for any complete lattice of annotations, but we only need it for sets. We can now define the collecting semantics as a least fixed-point:

$$\begin{aligned}
&\mathit{CS} :: \mathit{com} \Rightarrow \mathit{state\ set\ acom}\\
&\mathit{CS}\ c = \mathit{lfp}\ (\mathit{step}\ \mathit{UNIV})\ c
\end{aligned}$$

where *UNIV* is the set of all elements of a type, in this case the set of all states. That is, the set of initial states are all states. This is a standard choice but any other set is equally possible.

4.3 Small-Step Semantics

The collecting semantics can be specialized to a small-step semantics executing a command c starting in a state s: annotate c with $\emptyset$ everywhere, make a single step with initial state set $\{s\}$ (now s has been "injected" into c), but now keep stepping c with empty initial state set:

$$\mathit{steps}\ s\ c\ n = (\mathit{step}\ \emptyset)^n\ (\mathit{step}\ \{s\}\ (\mathit{anno}\ \emptyset\ c))$$

This describes $n{+}1$ steps of a small-step operational semantics. The resulting command will take one of two forms: either it is annotated with $\emptyset$ everywhere, which means that the execution terminated and the state has "dropped out" at the end; or it contains exactly one non-empty annotation, which is a singleton $\{s'\}$ that shows exactly where the execution currently is.

We can animate the small-step semantic just like the full collecting semantics, by evaluating *steps*. The output below is generated by executing

```
value show-acom ["x"] (steps (λx. 0) ss-ex n)
```

in Isabelle, for increasing n, which is very effective in class. The first 4 iterations produce the following output:

```
{{[("x", 0)]}}
WHILE Less (V "x") (N 1) DO "x" ::= Plus (V "x") (N 2) {∅}
{∅}

{∅}
WHILE Less (V "x") (N 1)
DO "x" ::= Plus (V "x") (N 2) {{[("x", 2)]}}
{∅}

{{[("x", 2)]}}
WHILE Less (V "x") (N 1) DO "x" ::= Plus (V "x") (N 2) {∅}
{∅}

{∅}
WHILE Less (V "x") (N 1) DO "x" ::= Plus (V "x") (N 2) {∅}
{{[("x", 2)]}}
```

One more step, and the single state drops out.

The whole point of this operational semantics is to justify the least-fixed point construction of *CS* with respect to it. More precisely, we show that *CS* overapproximates the operational semantics:

Lemma. $steps\ s\ c\ n \leq CS\ c$

The two semantics actually coincide, but we only need one direction. Later we show that the abstract interpreter overapproximates the collecting semantics. Together this proves that the abstract interpreter overapproximates the small-step semantics.

The above small-step semantics is rather non-standard (but attractively simple). Cachera and Pichardie [3] present a proof relating a standard small-step semantics to a collecting semantics. Their proof should carry over to our framework if their program points are simulated by our annotations.

5 Abstract Interpretation

This and the following two sections develop and refine a generic abstract interpreter. Initially, boolean expressions are not analysed. This is corrected in a second step. In a last step, widening and narrowing are added.

5.1 Orderings

The various orderings we need are defined as type classes. The notation $\tau :: C$ means that type τ is of class C.

A type $'a$ is a *preorder* ($'a$ :: *preord*) if there is a reflexive and transitive relation $\sqsubseteq$ :: $'a \Rightarrow {'a} \Rightarrow$ *bool*. We do not assume antisymmetry because we want to cover types with multiple different representations for the same abstract element, e.g. pairs as intervals, where all pairs (l, h) with $h < l$ represent the empty interval.

Any relation $\sqsubseteq$ on type $'a$ extends to type $'a$ *acom* exactly like $\leq$ in the definition of the collecting semantics in Section 4.

Lemma. If $'a$ :: *preord* then $'a$ *acom* :: *preord*.

In Isabelle, such lemmas are expressed as so-called instance statements. They allow the type checker to infer the class of complex types automatically.

Our abstract domains will initially be semilattices. Later we extend them to lattices. A type $'a$ is a *semilattice with top* ($'a$ :: *SL-top*) if it is a preorder and there is a least upper bound (join) operation $\sqcup$ :: $'a \Rightarrow {'a} \Rightarrow {'a}$, i.e.

$$x \sqsubseteq x \sqcup y \qquad y \sqsubseteq x \sqcup y \qquad [\![x \sqsubseteq z;\ y \sqsubseteq z]\!] \Longrightarrow x \sqcup y \sqsubseteq z$$

and there is a top element $\top$ :: $'a$, i.e. $x \sqsubseteq \top$.

Both *option* and function types preserve semilattices:

Lemma. If $'a$ :: *SL-top* then $'a$ *option* :: *SL-top*.

The extension adjoins *None* as the least element.

Lemma. If $'a$ :: *SL-top* then $'b \Rightarrow {'a}$ :: *SL-top*.

The orderings extends pointwise in the usual manner.

5.2 Abstract Interpretation with Functional Abstract States

We start with an abstract interpreter that operates on abstract states that are functions. It is not yet executable, but a first, conceptually simple design that is made executable in a second step.

The abstract interpreter is parameterized with a type $'av$ :: *SL-top* of abstract values that comes with a concretization function γ. In Isabelle this is expressed as a locale:

locale *Val-abs* =
fixes γ :: $'av$::*SL-top* $\Rightarrow$ *val set*
assumes $a \sqsubseteq b \Longrightarrow \gamma\ a \subseteq \gamma\ b$ **and** $\gamma\ \top = UNIV$

The **fixes** part declares the parameters, the **assumes** part states assumptions on the parameters. As explained in the introduction, we only model half the abstract interpretation theory: we drop the abstraction function α and do not calculate abstract interpreters from concrete ones but merely prove given abstract interpreters correct.

In the context of this locale we define abstract interpreters for *aexp* and *acom*. They operate on a lifted abstract state of type $'av$ *st option* where

$$'av\ st = vname \Rightarrow {'av}$$

Type *option* allows us to model unreachable program points by annotating them with *None*, the counterpart to $\emptyset$ in the collecting semantics.

The concretization function γ is extended to *'av option st acom* in the canonical manner, preserving monotonicity:

$$\gamma_f :: 'av\ st \Rightarrow state\ set$$
$$\gamma_f\ S = \{s \mid \forall x.\ s\ x \in \gamma\ (S\ x)\}$$

$$\gamma_o :: 'av\ st\ option \Rightarrow state\ set$$
$$\gamma_o\ None = \emptyset$$
$$\gamma_o\ (Some\ S) = \gamma_f\ S$$

$$\gamma_c :: 'av\ st\ option\ acom \Rightarrow state\ set\ acom$$
$$\gamma_c\ c = map\text{-}acom\ \gamma_o\ c$$

where *map-acom f c* applies *f* to all annotations in *c*.

Now we come to the actual interpreters. An abstraction of *aval* requires abstractions of the basic arithmetic operations. Hence locale *Val-abs* is actually richer than we pretended above: it contains abstractions of *N* and *Plus*, too:

fixes $num' :: val \Rightarrow 'av$
assumes $n \in \gamma\ (num'\ n)$
fixes $plus' :: 'av \Rightarrow 'av \Rightarrow 'av$
assumes $[\![n_1 \in \gamma\ a_1;\ n_2 \in \gamma\ a_2]\!] \Longrightarrow n_1 + n_2 \in \gamma\ (plus'\ a_1\ a_2)$

The abstract interpreter for *aexp* is standard

$$aval' :: aexp \Rightarrow 'av\ st \Rightarrow 'av$$

$$aval'\ (N\ n)\ S = num'\ n$$
$$aval'\ (V\ x)\ S = S\ x$$
$$aval'\ (Plus\ a_1\ a_2)\ S = plus'\ (aval'\ a_1\ S)\ (aval'\ a_2\ S)$$

and its correctness ($s \in \gamma_f\ S \Longrightarrow aval\ a\ s \in \gamma\ (aval'\ a\ S)$) is trivial.

The abstract interpreter for annotated commands is defined like the collecting semantics in two stages. We start with an abstraction of *step*, where the notation $f(x := y)$ is predefined and means function update:

$$step' :: 'av\ st\ option \Rightarrow 'av\ st\ option\ acom \Rightarrow 'av\ st\ option\ acom$$

$$step'\ S\ (SKIP\ \{P\}) = SKIP\ \{S\}$$
$$step'\ S\ (x ::= e\ \{P\})$$
$$= x ::= e\ \{\mathsf{case}\ S\ \mathsf{of}\ None \Rightarrow None \mid Some\ S \Rightarrow Some\ (S(x := aval'\ e\ S))\}$$
$$step'\ S\ (c_1;\ c_2) = step'\ S\ c_1;\ step'\ (post\ c_1)\ c_2$$
$$step'\ S\ (IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{P\})$$
$$= IF\ b\ THEN\ step'\ S\ c_1\ ELSE\ step'\ S\ c_2\ \{post\ c_1 \sqcup post\ c_2\}$$
$$step'\ S\ (\{Inv\}\ WHILE\ b\ DO\ c\ \{P\})$$
$$= \{S \sqcup post\ c\}\ WHILE\ b\ DO\ step'\ Inv\ c\ \{Inv\}$$

Correctness of *step'* wrt *step* is proved by induction on *c*:

Lemma. If $S \subseteq \gamma_o\ S'$ and $c \leq \gamma_c\ c'$ then $\mathit{step}\ S\ c \leq \gamma_c\ (\mathit{step}'\ S'\ c')$

The abstract interpreter is defined by fixed-point iteration of *step'*. This raises the termination question. Because proof assistants like Coq and Isabelle/HOL build on logics of total functions, previous formalizations (e.g. the work by Pichardie) built the termination requirement into the ordering $\sqsubseteq$. We define the iteration for arbitrary orderings and prove termination separately. The slight advantage in a teaching context is that it allows us to postpone the discussion of termination. Our trick is to use $\mathit{while\text{-}option} :: ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ \mathit{option}$ from the Isabelle/HOL library. It satisfies the recursion equation

$$\mathit{while\text{-}option}\ b\ c\ s = (\textsf{if}\ b\ s\ \textsf{then}\ \mathit{while\text{-}option}\ b\ c\ (c\ s)\ \textsf{else}\ \mathit{Some}\ s)$$

which makes it executable. Mathematically, $\mathit{while\text{-}option}\ b\ c\ s = \mathit{None}$ in case the recursion does not terminate. We define a generic post-fixed point finder

$$\mathit{pfp} :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ \mathit{option}$$
$$\mathit{pfp}\ f = \mathit{while\text{-}option}\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ f$$

and as a special case the abstract interpreter:

$$\mathit{AI} :: \mathit{com} \Rightarrow 'av\ \mathit{st}\ \mathit{option}\ \mathit{acom}\ \mathit{option}$$
$$\mathit{AI}\ c = \mathit{pfp}\ (\mathit{step}'\ \top)\ (\bot_c\ c)$$

where $\bot_c = \mathit{anno}\ \mathit{None}$ (note that $\bot_c$ is one symbol). Iteration starts with $\bot_c\ c$, the least annotated version of c, thus making sure we obtain the least post-fixed point (if f is monotone). This is nice to know, but not used later on: for correctness, any post-fixed point will do. We iterate $\mathit{step}'\ \top$, corresponding to *step UNIV* in the collecting semantics.

Theorem. (Correctness of *AI* wrt *CS*) $\mathit{AI}\ c = \mathit{Some}\ c' \Longrightarrow \mathit{CS}\ c \leq \gamma_c\ c'$

It follows essentially because *CS* is defined as the *least* (post-)fixed point, *AI* returns a post-fixed point, and *step'* and *step* operate in lock-step.

This is the initial version of our generic abstract interpreter. Unfortunately it is not executable: in each iteration of *pfp* we need to test if the old and the new version of the annotated command are related by $\sqsubseteq$. This in turn requires us to compare all annotations, which are (optional) functions. But $\sqsubseteq$ on functions is not computable if the domain is infinite, which *vname* is. Before we fix this, a remark on monotonicity.

So far, monotonicity at the abstract level has not entered the picture: it is not needed for correctness of the basic abstract interpreter but will be required for termination. We define an extension of locale *Val-abs* (locales are hierarchical) where we also assume monotonicity of the abstract operations

$$\textbf{assumes}\ [\![a_1 \sqsubseteq b_1;\ a_2 \sqsubseteq b_2]\!] \Longrightarrow \mathit{plus}'\ a_1\ a_2 \sqsubseteq \mathit{plus}'\ b_1\ b_2$$

and call this the *monotone framework*. In this framework we can prove monotonicity of *step'*:

Lemma. If $S \sqsubseteq S'$ and $c \sqsubseteq c'$ then $\mathit{step}'\ S\ c \sqsubseteq \mathit{step}'\ S'\ c'$.

5.3 Abstract Interpretation with Computable Abstract States

We replace $vname \Rightarrow {}'av$ by finite functions because the state only needs to record values of variables that actually occur in the command being analysed. We could parameterize our abstract interpreter wrt a type of finite functions [12], but since we do not intend to provide multiple implementations, we fix a particularly simple model and redefine $'a\ st$ as follows:

$$\textbf{datatype}\ {}'a\ st = FunDom\ (vname \Rightarrow {}'a)\ (vname\ list)$$

That is, we record the domain of the finite function as a list. The two projection functions are $fun\ (FunDom\ f\ xs) = f$ and $dom\ (FunDom\ f\ xs) = xs$. Function update is easy:

$$update\ F\ x\ y =$$
$$FunDom\ ((fun\ F)(x := y))\ (\mathsf{if}\ x \in set\ (dom\ F)\ \mathsf{then}\ dom\ F\ \mathsf{else}\ x{\cdot}dom\ F)$$

where *set* converts a list into a set and "·" is Cons. Function application is called *lookup* and requires $'a$ to have a $\top$ element which is returned outside the domain:

$$lookup\ F\ x = (\mathsf{if}\ x \in set\ (dom\ F)\ \mathsf{then}\ fun\ F\ x\ \mathsf{else}\ \top)$$

Why $\top$? This reflects that our analysis assumes that uninitialized variables can have arbitrary values.

Lemma. If $'a :: SL\text{-}top$ then $'a\ st :: SL\text{-}top$.

The ordering is again pointwise (but expressed with lookup). The join intersects the domains because outside the domain lookup returns $\top$.

The development of the abstract interpreter stays exactly the same, except that application and update on type $'av\ st$ are called *lookup* and *update*. We have arrived at our first executable abstract interpreter. The initial development in terms of abstract states as functions was merely presented for didactic reasons, to keep it as simple as possible and introduce improvements gradually.

In addition we also prove a generic termination theorem. It is phrased directly in terms of measures because this is most convenient for our applications. In the context of the monotone framework (see end of previous subsection) we obtain

Theorem. $\exists\, c'.\ AI\ c = Some\ c'$ if there is a measure $m :: {}'av \Rightarrow nat$ such that $x \sqsubseteq y \wedge \neg\ y \sqsubseteq x \longrightarrow m\ y < m\ x$ and $x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$.

The fact that $while\text{-}option\ b\ f\ x = Some\ y$ means termination follows from the recursion equation for *while-option* (see above) together with the fact that $while\text{-}option\ b\ f\ x = None$ in case $b\ (f^k\ x)$ for all k.

6 Backward Analysis of Boolean Expressions

So far we have not analyzed boolean expressions at all. Now we take them into account by defining an analysis that "filters" an abstract state S wrt some boolean expression b and some intended result r of b: the resulting abstract state

S' should be more precise than S, i.e. $\gamma_o\ S' \subseteq \gamma_o\ S$, but no state that makes b evaluate to r must be lost: if $s \in \gamma_o\ S$ and $bval\ b\ s = r$ then also $s \in \gamma_o\ S'$. This filtering of abstract states corresponds to an intersection and is realized by the dual of the join, the *meet*. We also need to model the situation that some variable has no possible value, which corresponds to a least abstract element $\bot$. Therefore we upgrade from a semilattice to a lattice. A type $'a$ is a *lattice with top and bottom* ($'a$ :: *L-top-bot*) if it is a semilattice with top and there is a greatest lower bound (meet) operation $\sqcap :: 'a \Rightarrow 'a \Rightarrow 'a$, i.e.

$$x \sqcap y \sqsubseteq x \qquad x \sqcap y \sqsubseteq y \qquad [\![x \sqsubseteq y;\ x \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq y \sqcap z$$

and there is a bottom element $\bot :: 'a$, i.e. $\bot \sqsubseteq x$.

We specialize the *Val-abs* interface further by requiring $'av$::*L-top-bot* and by adding two further assumptions:

assumes $\gamma\ a_1 \cap \gamma\ a_2 \subseteq \gamma\ (a_1 \sqcap a_2)$ **and** $\gamma\ \bot = \emptyset$

The first assumption actually implies $\gamma\ (a_1 \sqcap a_2) = \gamma\ a_1 \cap \gamma\ a_2$. Moreover we require abstract filter functions for all basic arithmetic and boolean operations:

fixes *test-num'* :: $int \Rightarrow 'av \Rightarrow bool$
fixes *filter-plus'* :: $'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$
fixes *filter-less'* :: $bool \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$
assumes *test-num'* $n\ a = (n \in \gamma\ a)$
assumes *filter-plus'* $a\ a_1\ a_2 = (b_1,\ b_2) \Longrightarrow$
$[\![\ n_1 \in \gamma\ a_1;\ n_2 \in \gamma\ a_2;\ n_1 + n_2 \in \gamma\ a\ ]\!] \Longrightarrow n_1 \in \gamma\ b_1 \wedge n_2 \in \gamma\ b_2$
assumes *filter-less'* $(n_1 < n_2)\ a_1\ a_2 = (b_1,\ b_2) \Longrightarrow$
$[\![\ n_1 \in \gamma\ a_1;\ n_2 \in \gamma\ a_2\ ]\!] \Longrightarrow n_1 \in \gamma\ b_1 \wedge n_2 \in \gamma\ b_2$

The filter functions are similar to inverse functions: but instead of computing the arguments from the result, they are given both the arguments and the result and should return the filtered arguments where values that cannot lead to the given result may be removed. The **assumes** clauses express it the other way around: the *filter-plus'* clause says that values in the conretization of a_1 and a_2 that lead into $\gamma\ a$ must not be filtered out. This assumptions guarantees soundness. Based on the basic filtering functions we can now filter wrt *aexp* and later *bexp* as explained in the introduction of this section:

```
afilter :: aexp ⇒ 'av ⇒ 'av st option ⇒ 'av st option

afilter (N n) a S = (if test-num' n a then S else None)
afilter (V x) a S =
(case S of None ⇒ None
 | Some S ⇒
    let a' = lookup S x ⊓ a
    in if a' ⊑ ⊥ then None else Some (update S x a'))
afilter (Plus e1 e2) a S =
(let (a1, a2) = filter-plus' a (aval'' e1 S) (aval'' e2 S)
 in afilter e1 a1 (afilter e2 a2 S))
```

where $aval''$ is just a lifted version of $aval'$:

$$aval''\ e\ None = \bot$$
$$aval''\ e\ (Some\ S) = aval'\ e\ S$$

Note that the test $a' \sqsubseteq \bot$ in the *afilter* $(V\ x)$ clause prevents an imprecision. We could always return $Some\ (update\ S\ x\ a')$, as some authors do [14]. But if a' is $\bot$, this is really an unreachable state. However, this information can be overwritten in subsequent assignments, and when the resulting state is joined with another execution path, e.g. at the end of a conditional, the unreachable state can lead to a loss of precision. Hence we avoid creating states with $\bot$ components and work with the least state *None* instead.

Filtering with *bexp* is similar:

```
$bfilter :: bexp \Rightarrow bool \Rightarrow 'av\ st\ option \Rightarrow 'av\ st\ option$

$bfilter\ (Bc\ v)\ res\ S = (\text{if}\ v = res\ \text{then}\ S\ \text{else}\ None)$
$bfilter\ (Not\ b)\ res\ S = bfilter\ b\ (\neg\ res)\ S$
$bfilter\ (And\ b_1\ b_2)\ res\ S =$
$(\text{if}\ res\ \text{then}\ bfilter\ b_1\ True\ (bfilter\ b_2\ True\ S)$
 $\text{else}\ bfilter\ b_1\ False\ S \sqcup bfilter\ b_2\ False\ S)$
$bfilter\ (Less\ e_1\ e_2)\ res\ S =$
$(\text{let}\ (res_1, res_2) = filter\text{-}less'\ res\ (aval''\ e_1\ S)\ (aval''\ e_2\ S)$
 $\text{in}\ afilter\ e_1\ res_1\ (afilter\ e_2\ res_2\ S))$
```

Note that the then-case in *bfilter* $(And\ b_1\ b_2)$ is a tricky way to express *bfilter* b_1 *True* $\sqcap$ *bfilter* b_2 *True*, thus obviating the need to define $\sqcap$ on abstract states. It is debatable if this trick is a good idea in a teaching context.

Two of the defining equations for $step'$ are now refined

```
$step'\ S\ (IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{P\}) =$

$IF\ b\ THEN\ step'\ (bfilter\ b\ True\ S)\ c_1\ ELSE\ step'\ (bfilter\ b\ False\ S)\ c_2$
$\{post\ c_1 \sqcup post\ c_2\}$

$step'\ S\ (\{Inv\}\ WHILE\ b\ DO\ c\ \{P\}) =$

$\{S \sqcup post\ c\}$
$WHILE\ b\ DO\ step'\ (bfilter\ b\ True\ Inv)\ c$
$\{bfilter\ b\ False\ Inv\}$
```

but the definition of the abstract interpreter AI itself is unchanged. The correctness proof stays largely the same but requires two new lemmas:

Lemma. If $s \in \gamma_o\ S$ and $aval\ e\ s \in \gamma\ a$ then $s \in \gamma_o\ (afilter\ e\ a\ S)$.

Lemma. If $s \in \gamma_o\ S$ then $s \in \gamma_o\ (bfilter\ b\ (bval\ b\ s)\ S)$.

7 Widening and Narrowing

Widening is meant to ensure termination of fixed point iteration even in lattices of infinite height, eg intervals. More generally, it is meant to accelate convergence. Instead of computing $f^i(\bot)$ for $i = 0, 1, \ldots$ until a post-fixed point is found (see *pfp*), widening allows us to take bigger steps thus avoiding nontermination. These bigger steps may lose precision. Narrowing, another iteration, is meant to regain it.

A widening operator ∇ has type $'a \Rightarrow 'a \Rightarrow 'a$ and satisfies $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$. A narrowing operator $\triangle$ has type $'a \Rightarrow 'a \Rightarrow 'a$ and satisfies $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \triangle y$ and $y \sqsubseteq x \Longrightarrow x \triangle y \sqsubseteq x$. For convenience we put both of them in class *WN* and make it a subclass of *SL-top*.

Normally the axioms of widening and narrowing also include an ascending chain condition. We have again chosen to separate the termination argument. (Strictly speaking, widening would not need any axioms for correctness but only for termination.) Both operators can be extended to type *option* and *st*:

Lemma. If $'a$ :: *WN* then $'a$ *st* :: *WN*.

Lemma. If $'a$ :: *WN* then $'a$ *option* :: *WN*.

For the didactic reason of simplicity we have chosen not to apply widening or narrowing selectively at individual annotations but simultaneously everywhere. This can be less precise than more selective strategies [3] but is much simpler.

We define a function *map2-acom* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom$ that applies a function simultaneously to the corresponding annotations of two *strip*-equal annotated commands. This permits us to lift ∇ and $\triangle$ to ∇_c and $\triangle_c$ on annotated commands: $c_1 \nabla_c c_2 = map2\text{-}acom\ (op\ \nabla)\ c_1\ c_2$ and $c_1 \triangle_c c_2 = map2\text{-}acom\ (op\ \triangle)\ c_1\ c_2$, where $(op \bowtie)$ is the function some infix operator $\bowtie$ stands for.

Iterative widening and narrowing on *acom* are expressed as loops:

$$iter\text{-}widen\ f = while\text{-}option\ (\lambda c.\ \neg\ f\ c \sqsubseteq c)\ (\lambda c.\ c \nabla_c f\ c)$$
$$iter\text{-}narrow\ f = while\text{-}option\ (\lambda c.\ \neg\ c \sqsubseteq c \triangle_c f\ c)\ (\lambda c.\ c \triangle_c f\ c)$$

This formalizes one of the widening variants proposed by Cousot [6, footnote 6]. Pichardie and Monniaux [7] propose other formalizations.

The overall analysis performs widening first and then narrowing:

$$pfp\text{-}wn\ f\ c =$$
$$(\mathsf{case}\ iter\text{-}widen\ f\ (\bot_c\ c)\ \mathsf{of}\ None \Rightarrow None \mid Some\ c' \Rightarrow iter\text{-}narrow\ f\ c')$$

Later we show that the *None* case cannot arise under certain assumptions about widening. By definition, *iter-widen* $f\ (\bot_c\ c)$ finds a post-fixed point c' of f if it terminates. Assuming f is monotone, induction together with the narrowing properties shows that *iter-narrow* $f\ c'$ finds another post-fixed point of f below c' if it terminates.

In the context of the monotone framework we define *AI-wn* with the help of *pfp-wn* instead of *pfp*, as previously:

$$AI\text{-}wn = pfp\text{-}wn\ (step'\ \top)$$

The correctness ($AI\text{-}wn\ c = Some\ c' \Longrightarrow CS\ c \le \gamma_c\ c'$) proof is as before.

7.1 Termination

Correctness of widening and narrowing was easy. Termination is quite technical, although we have adopted an approach that does not refer to infinite chains but is phrased in terms of measure functions. For widening, each type needs to come with a measure function m into *nat* such that

$$x \sqsubseteq y \Longrightarrow m\ y \leq m\ x$$
$$\neg\ y \sqsubseteq x \Longrightarrow m\ (x \bigtriangledown y) < m\ x$$

The first measure property guarantees that the measure cannot go up with a widening step: the first widening axiom implies $m\ (x \bigtriangledown y) \leq m\ x$ (the second widening axiom is never needed). The second measure property guarantees that with every widening step of *iter-widen*, the measure goes down. The second property is the one we need, the first one is only auxiliary.

Both measure properties together allow us to lift them to composite data types, especially abstract states and annotated commands. Both types are just glorified tuples and hence we can explain the mechanism in terms of pairs without having to bother with the technical details of the more complex types. Everything on pairs is defined componentwise, including the measure function and the function f whose post-fixed point we seek:

$$((y_1,\ y_2) \sqsubseteq (x_1,\ x_2)) = (y_1 \sqsubseteq x_1 \wedge y_2 \sqsubseteq x_2)$$
$$(x_1,\ x_2) \bigtriangledown (y_1,\ y_2) = (x_1 \bigtriangledown y_1,\ x_2 \bigtriangledown y_2)$$
$$m\ (x_1,\ x_2) = m_1\ x_1 + m_2\ x_2$$
$$f\ (x_1,\ x_2) = (f_1\ x_1, f_2\ x_2)$$

The first measure property, anti-monotonicity, lifts trivially to pairs. Let us now consider the second measure property and assume $\neg\ f\ (x_1,\ x_2) \sqsubseteq (x_1,\ x_2)$, i.e. either $\neg\ f_1\ x_1 \sqsubseteq x_1$ or $\neg\ f_2\ x_2 \sqsubseteq x_2$. In the first case we have $m_1(x_1 \bigtriangledown f_1\ x_1) < m_1\ x_1$ (by the second measure property) and $m_2(x_2 \bigtriangledown f_2\ x_2) \leq m_2\ x_2$ (by the first measure property) and thus $m\ ((x_1,\ x_2) \bigtriangledown f\ (x_1,\ x_2)) = m_1\ (x_1 \bigtriangledown f_1\ x_1) + m_2\ (x_2 \bigtriangledown f_2\ x_2) < m_1\ x_1 + m_2\ x_2$. The second case is dual.

This way we can lift the two measure properties from the basic domain of abstract values up to annotated commands. However, there are some technicalities. The x and y in the measure properties need to fulfill additional invariants, in particular at the *acom* level: both must be *strip*-equal annotated commands over the same fixed finite set of variables. Hence the full measure theorem becomes

If *finite* X, *strip* c' = *strip* c, $c \in$ *Com* X, $c' \in$ *Com* X and $\neg\ c' \sqsubseteq c$, then $m\ (c \bigtriangledown_c c') < m\ c$.

where m is the measure function on *acom* and *Com* X is the set of commands whose annotations mention only variables in X. Of course *step'* preserves these invariants.

Termination of narrowing is proved in a similar manner, using measure functions called n that must also satisfy two properties:

$$x \sqsubseteq y \Longrightarrow n\ x \leq n\ y$$
$$y \sqsubseteq x \Longrightarrow \neg\ x \sqsubseteq x \bigtriangleup y \Longrightarrow n(x \bigtriangleup y) < n\ x$$

Again, the first property lifts trivially but it is the second one we are really after. It is lifted to pairs in a similar manner as for widening, using the second narrowing axiom. Obtaining the final measure theorem for narrowing on the *acom* level is again technical in the same way as for widening. At the end of the day, here is the unconditional termination statement for *AI-ivl′*, the instantiation of *AI-wn* with intervals:

Theorem $\exists\, c'.\ \textit{AI-ivl}'\ c = \textit{Some}\ c'$

7.2 Intervals

We have instantiated the various frameworks above with the standard analyses, in particular intervals. Our definition of intervals is extremely basic:

datatype *ivl* = *I* (*int option*) (*int option*)

where *None* represents infinity. For readability we install some syntactic sugar: $\{i\ldots j\}$ stands for *I* (*Some i*) (*Some j*); infinite lower or upper bounds are simply dropped. For example, $\{i\ldots\}$ is *I* (*Some i*) *None*. The only drawback is that the empty interval has many representations, but this is why our value abstraction is based on preorders, not partial orders. We refrain from giving the details of the operations on intervals. They follow the literature, except for the representation.

Just like for the small-step semantics, we can animate the computation of the abstract interpreter by iterating the step function and widening/narrowing. We evaluate *show-acom* $((\lambda c.\ c\ \nabla_c\ \textit{step-ivl}\ \top\ c)^n\ (\bot_c\ \textit{testc}))$ for increasing n. The pretty-printing function *show-acom* shows an abstract state as a list of pairs (*x*,*ivl*) — no need to supply the list of variables, it is part of the abstract state.

For $n = 1$ we obtain the program annotated with *None* everywhere except after the first assignment:

```
''x'' ::= N 7 {Some [(''x'', {7...7})]};
{None}
WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 1) {None}
{None}
```

The next step merely initializes the invariant:

```
''x'' ::= N 7 {Some [(''x'', {7...7})]};
{Some [(''x'', {7...7})]}
WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 1) {None}
{None}
```

Now the invariant filtered with the loop condition is propagated to the end of the loop body:

```
''x'' ::= N 7 {Some [(''x'', {7...7})]};
{Some [(''x'', {7...7})]}
WHILE Less (V ''x'') (N 100)
DO ''x'' ::= Plus (V ''x'') (N 1) {Some [(''x'', {8...8})]}
{None}
```

In the next step, widening has an effect and combines $\{7\ldots7\}$ and $\{8\ldots8\}$ into the new invariant $\{7\ldots\}$:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7\ldots7\})]\};$
$\{Some\ [(''x'', \{7\ldots\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8\ldots8\})]\}$
$\{None\}$

One more iteration yields a (post-)fixed point of *step-ivl*:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7\ldots7\})]\};$
$\{Some\ [(''x'', \{7\ldots\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8\ldots\})]\}$
$\{Some\ [(''x'', \{100\ldots\})]\}$

Switching to narrowing now, we obtain a second (post-)fixed point of *step-ivl* after 3 more iterations:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7\ldots7\})]\};$
$\{Some\ [(''x'', \{7\ldots100\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8\ldots100\})]\}$
$\{Some\ [(''x'', \{100\ldots100\})]\}$

8 Conclusion

The above material was covered in 4 weeks (with two 90 minutes lectures per week) in a 15 weeks MSc course on semantics via a theorem prover. Much of it worked well, although a few points are still a bit technical. In particular, we did not cover termination formally, especially for widening/narrowing. We intend to streamline this issue further in the future.

The Isabelle theories are available online at `http://isabelle.in.tum.de/library/HOL/HOL-IMP/` (the relevant theories are named `*ITP`) and in the Isabelle distribution in `src/HOL/IMP/Abs_Int_ITP/`.

Acknowledgement. David Pichardie's exemplary review and his many explanations greatly improved my understanding of his work and of abstract interpretation in general. Brian Huffman's comments improved the presentation.

References

1. Bertot, Y.: Structural Abstract Interpretation: A Formal Study Using COQ. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) Language Engineering and Rigorous Software Development. LNCS, vol. 5520, pp. 153–194. Springer, Heidelberg (2009)

2. Bertot, Y., Grégoire, B., Leroy, X.: A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 66–81. Springer, Heidelberg (2006)
3. Cachera, D., Pichardie, D.: A Certified Denotational Abstract Interpreter. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 9–24. Springer, Heidelberg (2010)
4. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, Steinbrüggen (eds.) Calculational System Design. IOS Press (1999)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM Symp. Principles of Programming Languages, pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
7. Monniaux, D.: A minimalistic look at widening operators. Higher-Order and Symbolic Computation 22, 145–154 (2009)
8. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
9. Nipkow, T.: Verified Bytecode Verifiers. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 347–363. Springer, Heidelberg (2001)
10. Nipkow, T.: Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 24–38. Springer, Heidelberg (2012)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Pichardie, D.: Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés. Ph.D. thesis, Université Rennes 1 (2005)
13. Pichardie, D.: Building certified static analysers by modular construction of well-founded lattices. In: Proc. 1st International Conference on Foundations of Informatics, Computing and Software (FICS 2008). ENTCS, vol. 212, pp. 225–239 (2008)
14. Seo, S., Yang, H., Yi, K.: Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 230–245. Springer, Heidelberg (2003)

Verifying and Generating WP Transformers for Procedures on Complex Data

Patrick Michel and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{p_michel,poetzsch}@cs.uni-kl.de

Abstract. We present the formalized theory of a weakest precondition calculus for procedures on complex data with integrity constraints. The theory defines the assertion language and the wp-transformer. It contains the proofs for soundness and "weakestness" of the preconditions. Furthermore, we formalize a normalization process that eliminates all elementary updates from preconditions. This *normalization* property is important for *efficient checking* of the preconditions in programs. The theory is completely realized in Isabelle/HOL and used for generating the Haskell implementation of the wp-transformer and the normalization process.

The wp generation is developed for procedures on complex data with integrity constraints, for example XML documents satisfying a schema. Efficient checkability allows maintaining the constraints with acceptable computing resources. It is a central motivation of our work and has influenced many design decisions.

Keywords: Interactive verification, Isabelle/HOL, structured data, paths, invariants, precondition generation, language semantics.

1 Introduction

We present the formalized theory of a weakest precondition calculus for procedures on complex data with integrity constraints. The theory defines the assertion language and the wp-transformer. It contains the proofs for soundness and "weakestness" of the preconditions. Furthermore, we formalize a normalization process that eliminates all elementary updates from preconditions. This *normalization* property is important for *efficient checking* of the preconditions in programs. The theory is completely realized in Isabelle/HOL and used for generating the Haskell implementation of the wp-transformer and the normalization process.

Our main application area are procedures on complex structured data with integrity constraints, for example XML documents satisfying a schema. Such data is used in many different areas of computing, for example as objects satisfying a class invariant, as parameters of complex types used in web services, or as data stored in a database. Accordingly, the data should only be manipulated by procedures pc maintaining the integrity constraints. That is, the constraints play the role of data invariants C_{inv}. Each procedure consists of a sequence of elementary modifications of the data where the invariant might be violated in between. The goal is to compute preconditions $C_{pre}(pc)$ of these procedures, i.e., assertions ranging over the current data state and the parameters of pc, with the following properties:

1. If $C_{pre}(pc) \wedge C_{inv}$ holds in the prestate of pc, then C_{inv} holds in the poststate.
2. $C_{pre}(pc)$ can be checked efficiently in a given prestate.

Efficient checkability has two aspects. First, $C_{pre}(pc)$ should avoid existential quantifiers and only use read operations on the complex data. Second, $C_{pre}(pc)$ should not express properties that are covered by the invariant, because this would lead to redundant checking. These properties allow us to efficiently maintain even complex and large invariants by only checking $C_{pre}(pc)$ whenever pc is called (cf. Sect. 2).

Related to our work are wp-transformers for programs with heap allocated data [12, 17, 8] and language settings designed for XML or tree updates [11, 2, 18, 3]. Whereas these approaches either result in *inefficient* preconditions or work only for *structural* constraints, we aim to generate efficiently checkable preconditions for complex data with integrity constraints going beyond structure. In particular, preconditions should be normalized such that they only read the data in the prestate and do not contain updates. The central contribution of this paper is a theory for a core wp-calculus satisfying the described requirements. It contains:

– A path-based representation of structured data together with a suitable assertion and update language
– A wp-transformer together with proofs that it is sound and produces weakest preconditions
– A normalization process for assertions
– A theory for infinite multisets and a three-valued logic as semantical foundation for the assertion language, the transformer, and the normalization
– A proof technique and concept for syntactic transformations under semantic equivalence, using explicit partiality and a concept of safe formulas

The theory is formalized in Isabelle/HOL [16] (over 9000 lines) and is used to generate Haskell programs for the wp-transformer and the normalization (about 2200 lines of generated Haskell code). This tool-based approach proved to be indispensable, in particular for the rather complex normalization process.

Overview. Section 2 explains our approach to complex data in more detail. Section 3 presents the data representation and the core assertion language. Sections 4 to 6 summarize and discuss important aspects of the wp-theory. Section 7 discusses related work and Sect. 8 concludes.

2 Approach

In this section, we describe and motivate our approach by two small examples. The first shows how the different aspects of the approach work together, focusing on preconditions. The second is about a more realistic data type. To illustrate that our approach is not tied to a specific language setting we use a programming language syntax for the first example and an XML-related syntax for the second.

Preconditions. By *complex data*, we refer to hierarchical data structures with integrity constraints going beyond purely structural schemas. As first example, we consider a record type `container` with two components: the component `items` is an array of numbers storing for item i its weight; the component `owght` stores the overall sum of the weights of all items. As invariant of `containers` c, we have:

$$C_{inv} \;\equiv\; sum(c.\text{items}) = c.\text{owght} \;\wedge\; c.\text{items}[i] > 0 \;\wedge\; c.\text{owght} < 1000$$

where i quantifies over the defined indices of the array. A basic incremental procedure on containers could, for instance, modify the weight of an item:

```
PROC modifyWeight(container c, int ix, int wght)
  c.owght = c.owght - c.items[ix] + wght;
  c.items[ix] = wght;
```

Using the techniques from [12], the generated weakest precondition for C_{inv} is

$$\begin{array}{l} sum(update(c.\text{items},\, ix,\, wght)) = c.\text{owght} - c.\text{items}[ix] + wght \;\wedge \\ update(c.\text{items},\, ix,\, wght)[i] > 0 \;\wedge\; c.\text{owght} - c.\text{items}[ix] + wght < 1000 \end{array}$$

where *update* describes an array update and where we ignore index out of bounds problems for simplicity. Even for this simple example, the generated precondition is not suitable for efficient checking. The update-operation causes unneccessary overhead and the precondition essentially forces us to check the invariant C_{inv} for the prestate although we may assume that it holds. A much nicer precondition avoiding updates and rechecking of the invariant would be:

$$wght > 0 \;\wedge\; c.\text{owght} - c.\text{items}[ix] + wght < 1000$$

Generating such preconditions for efficient checking is the central goal of wp-transformation with normalization. Our normalization technique eliminates all update-operations. Furthermore, in the normalized form, the invariant can be factored out from the precondition using regrouping and simplification. We focus on loop-free, atomic procedures for incremental updates on complex data. For details on this design decision and how our approach can be used in practice see [14, 15].

Schemas for Complex Data. To illustrate the kinds of complex data that our approach supports, we consider an extension of the container datatype above. We use a schema notation in the style of the XML schema language RelaxNG [10]. The (extended) container type has a capacity attribute, contains a collection of items and a collection of products. Collections are multisets of elements (or attributes), i.e., have unordered content; elements in collections are referenced by a *key* that has to be provided on entry. Each item in the extended container type is considered to be an instance of a specific product and refers to the product entry using the corresponding key. Products have a positive weight less or equal to the capacity of the container.

```
element container {
 attribute capacity { integer [. > 0] },
 [ ./capacity ≥ sum (./product[./item/productref]/weight)]

 element item * {
   attribute productref { key [ //product[.] ∈ $ ] }
  },
 element product * {
   attribute weight { integer [. > 0] [. ≤ //capacity] }
} }
```

To formulate constraints, we use path-based expressions. The dot '.' refers to the value of the current attribute or element. The '$'-sign refers to the overall document. For example, the constraint `//product[.]` ∈ $ of attribute `productref` states that there has to be a product with this key in the current document. The global constraint in line 3 enforces that the capacity of the container is larger or equal to the sum of the weights of all items; i.e., the expression `./product[./item/productref]/weight` represents the *multiset* of all paths obtained by placing a key at `./item/productref` into `./product[_]/weight`. A more detailed presentation of our data description approach is given in [15].

We used our approach to define the schema, integrity constraints, and procedures for the persistent data of a mid-size web application [13]. The definition of the schema and integrity constraints is about 100 lines long. We developed 57 basic procedures for the manipulation of the data.

3 Core Assertions

This section defines the syntax and semantics of our core assertion language for complex data structures. This language is the heart of our theory, as the choice of the core operators play a central role to achieve our goals. On the one hand, syntactic transformations and normalizations require operators with homomorphic and further semantical properties. On the other hand, to define useful invariants, to handle partiality of operators, to manage the wp-transformations, and to support certain normalization steps within the language, we need enough expressive power. The developed language is kind of a sweet spot between these partially conflicting requirements. It essentially follows the ideas of [5] (cf. Sect. 7). It is a core language in the sense that syntactical extensions and further operators having the required transformation properties can be added on top of the language.

Data representation in the assertion language is based on the concepts of paths and the idea that every basic element of a hierarchical data structure should be addressable by a unique path. A path is essentially a sequence of labels l, which can be thought of as element names in XML or attribute names in Java, etc. As inner nodes of a hierarchical data structure might represent collections, the path has to be complemented by a *key* for each step to select a unique element of the collection at each point. Following XML, we call a hierarchical data structure a *document*.

Definition 1 (Paths and Documents). *A **path** is a sequence of label-key pairs. The special key null is used for path steps which need no disambiguation. The sequence of labels of a path alone defines the so-called **kind** of the path. A **document** is a finite mapping from a prefixed-closed set of paths to values.*

Syntax and Semantics. Based on the concept of documents, we define the syntax of the assertion language as follows:

$$\begin{array}{rl}
\textbf{value expr.} & V ::= c \mid v \mid \$(P) \mid V_{SI} \odot V_I \mid V \oplus V \mid \mathit{sum}\ V_I \mid \mathit{size}\ V \mid \mathit{count}\ V\ V_S \mid \mathit{tally}\ T\ V \\
\textbf{path expr.} & P ::= \mathit{root} \mid P/l[V_K] \mid P/l \qquad \textbf{types}\ T ::= \mathit{key} \mid \mathit{int} \mid \mathit{string} \mid \mathit{unit} \\
\textbf{relations} & R ::= V_S = V_S \mid V_{SI} < V_{SI} \qquad \textbf{disjunctions}\ D ::= \mathit{false} \mid L \vee D \\
\textbf{literals} & L ::= R \mid \neg R \qquad \textbf{conjunctions}\ C ::= \mathit{true} \mid D \wedge C
\end{array}$$

Basic *value*s are of type *key*, *int*, *string*, or *unit* where *unit* is a singleton type with the constant (). Expressions and assertions B are interpreted with respect to an environment E, denoted by $\| B \|_E$. The environment assigns keys to the variables and a document to the \$-symbol that refers to the document underlying the assertion.[1]

Path expressions denote the root node, select children by label-key pairs, or select all children for a label. The latter is called a *kind step*, as it selects all children of this particular kind. The semantic domain of path expressions are (possibly infinite) multisets of paths. The semantic equations are as follows:

$$\begin{aligned}
\| \mathit{root} \|_E &= \mathit{root} \\
\| P/l[V_K] \|_E &= \{ p/l[k] \mid p \leftarrow \| P \|_E,\ k \leftarrow \| V_K \|_E\ ,\ k \in \mathit{Univ}(\mathit{key}) \} \\
\| P/l \|_E &= \{ p/l[k] \mid p \leftarrow \| P \|_E,\ k \leftarrow\ \mathit{Univ}(\mathit{key}) \}
\end{aligned}$$

where we denote the universe of keys by $\mathit{Univ}(\mathit{key})$ and use multiset comprehensions to denote all paths $p/l[k]$ where p and k are generated from multisets.

Basic value expressions are constants c, such as $1, -1, 0$ and *null*, and variables v for keys. Variables are implicitly universally quantified at top level. To handle single values and multisets of values uniformly, we identify single values with the singleton multisets containing exactly that value and vice versa. The read expression $\$(P)$ selects all values from the underlying document \$ that are stored at paths P:

$$\| \$(P) \|_E = \ \{ E(\$)(p) \mid p \leftarrow \| P \|_E,\ p \in \mathit{dom}\ E(\$) \}$$

On multisets, we provide four aggregate functions: $\mathit{sum}\ V_I$ returns the sum of all integer elements of V_I; $\mathit{size}\ V$ returns the number of elements in V; $\mathit{count}\ V\ V_S$ returns the number of occurrences of singleton V_S in V; $\mathit{tally}\ T\ V$ returns the number of occurrences of elements of type T in V. Furthermore, we support the scalar multiplication $\odot$ of a single integer with a multiset of integers and a union operation $\oplus$ for multisets with $\mathit{count}\ (V_1 \oplus V_2)\ V_S = \mathit{count}\ V_1\ V_S + \mathit{count}\ V_2\ V_S$.

The rest of the syntax defines boolean formulas in conjunctive normal form with the polymorphic equality of singleton values and the ordering relation on singleton integers. The constants *false* and *true* are used for empty disjunctions or empty conjunctions. To handle partiality, all semantic domains include the bottom element $\perp$. All operators except for $\vee$ and $\wedge$ are strict: $\vee$ evaluates to true if one of the operands evaluates to true; otherwise its evaluation is strict (similar for $\wedge$).

In summary, the assertion language allows us to formulate document properties. Every hierarchical document model, in which values are accessed using a path concept, can be supported by our approach. As the core syntax does not support document updates, checking whether an assertion holds for a given document only needs to read values in the document by following access paths.

Syntactic Extensions. We extended the core language in two ways. Firstly, we support assertions that are not in conjunctive normal form. These more general forms are automatically transformed back into the core syntax. Secondly, we added further operators and abbreviations. We have taken great care in the design of the operators and their theory to avoid large blow-ups of the formula size as a result of eliminating the new operators (for details, we refer to the accompanying Isabelle/HOL theory).

[1] The document corresponds to the state of the heap in assertions for programming languages.

In particular, we provide the following abbreviations and operators:

$$\begin{array}{rcl@{\qquad}rcl}
\{\} & \equiv & 0 \odot \mathit{null} & \mathit{empty}\ V & \equiv & \mathit{size}\ V = 0 \\
\bot & \equiv & \mathit{null} \odot 0 & \mathit{unique}\ V & \equiv & \mathit{size}\ V = 1 \\
-V & \equiv & -1 \odot V & \mathit{unique}\ P & \equiv & \ldots \qquad \mathit{count}\ P_2\ P_1 \equiv \ldots \\
V_1 + V_2 & \equiv & \mathit{sum}\ (V_1 \oplus V_2) & P_1 \in P_2 & \equiv & \mathit{count}\ P_2\ P_1 > 0 \\
V_1 - V_2 & \equiv & V_1 + (-V_2) & P_1 \notin P_2 & \equiv & \mathit{count}\ P_2\ P_1 = 0 \\
V\ \mathit{is}\ T & \equiv & \mathit{tally}\ T\ V = \mathit{size}\ V & P \in \$ & \equiv & \mathit{unique}\ P \wedge \mathit{unique}\ \$(P) \\
V\ \mathit{is}-\mathit{not}\ T & \equiv & \mathit{tally}\ T\ V < \mathit{size}\ V & P \notin \$ & \equiv & \mathit{unique}\ P \wedge \mathit{empty}\ \$(P)
\end{array}$$

The left column shows simple abbreviations for the empty multiset, bottom, an embedding of integer arithmetic, and type tests. The right column adds some predicates for value multisets, as well as containment relations for paths regarding path multisets and the document. $\mathit{unique}(P)$ yields true if P evaluates to a singleton path; count denotes the count-operator on path expression (both operators are realized by syntactic transformation into a value expression of the core syntax).

4 WP-Transformer for Linear Programs

As explained in Sect. 2, the basic goal of our approach is to generate preconditions for procedures manipulating complex data or documents. The preconditions should be efficiently checkable and should guarantee that integrity constraints are maintained. This section defines the imperative language for document manipulation and the weakest precondition generation.

Language for Document Manipulation. Procedures have a name, a list of parameter declarations, and a statement of the following form as body:

$$\begin{array}{rl}
\textbf{statements}\ S ::= & \mathit{skip} \mid S;S \mid p := V_S \mid \mathit{if}\ L\ \mathit{then}\ S\ \mathit{else}\ S\ \mathit{fi} \mid \mathit{assert}\ C \mid \\
& \mathit{insert}\ P_S\ l\ V_{SK} \mid \mathit{update}\ P_S\ V_S \mid \mathit{delete}\ P_S
\end{array}$$

The skip-statement and sequential statement composition are as usual. The assignment has a local variable or parameter on the left-hand side and a (singleton) value expression free of (logical) variables as right-hand side. The boolean expression in the if-statement is restricted to literals. The assert-statement uses arbitrary assertions and allows strengthening of the precondition which can simplify it. There are three basic update-operations for documents:

- $\mathit{insert}\ P_S\ l\ V_{SK}$ assumes that the singleton path P_S exists in \$, but not the singleton path $P_S/l[V_{SK}]$; it inserts path $P_S/l[V_{SK}]$ with default value ().
- $\mathit{update}\ P_S\ V_S$ changes the value at an existing singleton path P_S to the singleton value V_S. In combination with *insert*, it allows us to insert arbitrary paths and values into a document.
- $\mathit{delete}\ P_S$ removes all values in the document associated with the existing singleton path P_S or any of its descendants.

We do not provide a loop-statement, because we want the wp-generation to be fully automatic and because loops are available in programs in which the procedures for document manipulation are called. The statements are designed (and proven) in such a way that they maintain the property that the set of paths in a document is prefix-closed. The semantics of statements is defined in big-step operational style:

Definition 2 (Semantics of Statements). *The semantics of statements is inductively defined as a judgment* $S, E_1 \rightsquigarrow E_2$*. The judgment holds if the execution of statement* S *starting in environment* E_1 *terminates, without errors, with environment* E_2*. The environments capture the state of the document, the parameters of the procedure, and the local program variables.*

Based on the semantics, we define Hoare-Triples for *total correctness*, i.e., if the precondition holds, no errors occur and a poststate exists. We have proven that statement execution is deterministic, i.e., the post environment is unique, if it exists.

Definition 3 (Hoare-Triple). *Hoare-Triples* $\{C\}\ S\ \{C\}$ *have the following semantics:*

$$\vDash \{C_P\}\ S\ \{C_Q\} \quad \equiv \quad \forall E.\ \|\ C_P\ \|_E\ \rightarrow\ \exists E'.\ S, E \rightsquigarrow E'\ \wedge\ \|\ C_Q\ \|_{E'}$$

WP Generation. The central goal of WP generation is to take an assertion C_Q and a statement S and generate the weakest assertion C_P such that $\vDash \{C_P\}\ S\ \{C_Q\}$. Unfortunately, the restriction of the assertion language to read operations in documents that is profitable for efficient checking has its downsides when it comes to wp generation. The classical approach to handle compound data in wp-calculi is to move updates in the programming language into updates in the assertions (as demonstrated by array c.items in Sect. 2). We solve this problem in two steps: First, we make the assertion language more expressive by adding document update operators such that wp generation can be expressed following the classical approach. Second, we show how these operators can be eliminated by a normalization process (see Sect. 6).

Besides document updates, we introduce program variables p (parameters, local variables) into the value expressions:

$$\begin{array}{lll} \textbf{value expr.} & V & ::= c \mid v \mid p \mid M(P) \mid \ldots \\ \textbf{documents} & M & ::= \$ \mid M[P_S \mapsto V_S] \mid M[P_S \mapsto] \end{array}$$

$M[P_S \mapsto V_S]$ combines insert and update: If $P_S \in dom\ M$, the value at P_S is changed; otherwise a new binding is created. $M[P_S \mapsto]$ deletes all bindings of P_S and all its descendants. With these additions, the wp-transformer can be defined as follows.

Definition 4 (WP-Transformer)

$$\begin{array}{ll}
wp\ skip\ C & = C \\
wp\ (S_1; S_2)\ C & = wp\ S_1\ (wp\ S_2\ C) \\
wp\ (delete\ P)\ C & = P \in \$\ \wedge\ C[\$/\$[P \mapsto]] \\
wp\ (insert\ P\ l\ V)\ C & = P \in \$\ \wedge\ C[\$/\$[P/l[V] \mapsto ()]]\ \wedge\ P/l[V] \notin \$ \\
 & \quad \wedge\ unique\ V\ \wedge\ V\ is\ key \\
wp\ (update\ P\ V)\ C & = P \in \$\ \wedge\ C[\$/\$[P \mapsto V]]\ \wedge\ unique\ V \\
wp\ (if\ L\ then\ S_1\ else\ S_2\ fi)\ C & = (L \rightarrow wp\ S_1\ C)\ \wedge\ (\neg L \rightarrow wp\ S_2\ C)\ \wedge\ (L \vee \neg L) \\
wp\ (p := V)\ C & = C[p/V] \\
wp\ (assert\ C_a)\ C & = C \wedge C_a
\end{array}$$

The notation $C[a/b]$ defines the substitution of all occurrences of a in C with b. The third conjunct in the precondition of the conditional is needed to make sure that evaluation of L does not lead to an error.

For the normalization, it is important to note how the data invariant, i.e., the postcondition of the procedure is changed by the transformer. As it does not refer to parameters or local variables, it is only modified at the document variable $ and gets augmented by new conjuncts. As core results for the wp-transformer, we have formally proven that the generated precondition is sound and is "weakest":

Theorem 1 (WP Sound and Weakest)

1. $\models \{wp\ S\ C\}\ S\ \{C\}$

2. $\models \{C_P\}\ S\ \{C_Q\}\ \wedge\ \|\ C_P\ \|_E\ =\ true\quad \Longrightarrow\ \|\ wp\ S\ C_Q\ \|_E = \ true$

5 Aspects of the Theory Formalization

In this section we discuss some of the techniques used to formalize the presented theory. Furthermore, we explain several of our design decisions, partly triggered by limitations of Isabelle/HOL, and introduce concepts needed for normalization.

Formalizing Semantic Domains. The lack of multi parameter type classes in Isabelle/HOL made it impossible to define a function symbol for the semantics, which is both polymorphic in the syntactic element and its semantic domain. To circumvent this, we defined a union type named *domain*, containing all semantic domains, as well as the bottom value. This makes the handling of bottom easier, because all semantic domains literally share the same bottom value. On the other hand, we had to prove that the semantics of a syntactic element is indeed either from its semantic domain or bottom. The resulting theorems in their different flavors (intro, elim and dest) are used extensively in the proofs of semantics related theorems.

Value, Path and Document Semantics. As the semantics of path expressions can be infinite path multisets, we developed a theory of infinite multisets.[2] More precisely, the multisets can contain infinitely many different values, but each value has only finitely many occurrences, i.e., the number of occurrences is still based on a function $\alpha \rightarrow$ nat. This decision allows to give a natural semantics to operators like kind steps and reads, yet it also makes dealing with multisets a bit more complicated in general. As an example consider the semantics of an operation *map* mapping a function f over all elements of a multiset and returning the multiset of the results. For instance, if f is constant for all elements, like the scalar multiplication with zero, the result of the mapping might not be covered by our multiset definition. Consequently, the definition and application of *map* has to take special care for such situations. However, in our framework, such a multiset theory is a good tradeoff, as it simplifies the treatment of paths which might lead to infinite multisets and gets restricted to finitely many values if used for value expressions.

The semantic domain of documents is based on the theory of partial functions as provided by the `Map` theory from Isabelle/HOL.

Mapping and Folding. With the abstract syntax defined for expressions, assertions, and statements, we want both to be able to make syntactic transformations and express properties. In a functional setting – and especially in Isabelle/HOL – a transformation is a bottom-up mapping of a family of functions onto the mutually recursive

[2] The multiset theory shipped with Isabelle/HOL is for finite multisets only.

datatypes. Bottom-up is important in Isabelle/HOL to guarantee termination. The family of functions has to contain an adequate function for every type involved, in our case P, V, M, L, R, D and C. We define the mapping function in a type class, such that it is polymorphic in the syntactic element.

A property on the other hand is described by a folding with a family of functions. We use the single parameter of the type class to make the function polymorphic in the type being folded, but we have to fix it with regard to the type being folded to. We are mostly interested in properties, i.e. boolean foldings, but also counting functions to support proving properties, i.e. nat foldings.

What we end up with are three functions *xmap*, *xprop* and *xcount*, which all take an adequate family of functions and then map or fold all the abstract syntax types we presented. We also defined instances for statements, so we can use the same properties on programs and their contained assertions and expressions.

Well-Formed Assertions. In the design of the assertion language, we decided for a very simple type system. In particular, we do not distinguish between singleton multisets and single values and paths. The main reason for this was to keep the assertion syntax as concise as possible and to avoid the introduction of similar operators for both cases. We can also avoid to deal with partiality from operators like read. Where we need the uniqueness property that an expression yields at most one value or path, we enforce it by checking expressions or assertions for well-formedness:

Definition 5 (Well-Formed Expressions and Assertions). *An expression is* ***statically unique****, if it denotes at most one singleton value in all environments. An expression or assertion is* ***well-formed****, if parameters of operators, whose semantics require a singleton value, are statically unique.*

On the core syntax, well-formedness can be checked based on the following criterion: all relations, as well as one of the parameters each in the count-operation and the scalar multiplication, must not directly contain the multiset plus or a read operation based on a path expression using kind steps. As paths and values form a mutual recursion, the property is also mutually recursive and excludes many more combinations. Once we introduce more operators to the syntax, the condition therefore gets more restrictive, yet we maintain the property at every step, so we can use it in the central steps of the normalization.

Safety. The main reason to include bottom into the semantics and use a three-valued logic was to handle partiality, i.e., exceptional behavior within the logic. It allows us to define the *domain* of assertions and expressions as the set of environments in which their semantics is not bottom. And, as we can use assertions as guards to widen the domains of the guarded assertions, we can construct assertions that are always defined. Such assertions are called *safe*. A function (or family of functions) f is considered to be *a domain identity* $id_E\, f$, if it does not change the semantics of an argument x within the domain of x.

Definition 6 (Safety and Domain Identities)

$$\mathit{safe}\ B \equiv \forall E.\ \| B \|_E \neq \bot \qquad \mathit{id}_E\ f \equiv \| B \|_E \neq \bot \longrightarrow \| f\ B \|_E = \| B \|_E$$

Having such a notion of failure and safety is beneficial for syntactic transformations and proving that such transformations are sound. Most transformations simply do not preserve semantics in all corner cases, as they might widen the domain of expressions. Using the notion of safety, we are able to prove, that *xmap* does not alter the semantics of a safe formula, if it uses a family of domain identities.

Theorem 2 (Safe Transformations)

$$\mathit{id}_E\ f \wedge \mathit{safe}\ C \implies \| \mathit{xmap}\ f\ C \|_E = \| C \|_E$$

6 Normalization

The goal of this section is to eliminate update-operations from generated preconditions and to get preconditions into a form where we can identify unchanged parts of the original invariant and separate them from new conjuncts. To reach this point, we need multiple, increasingly complex steps.

The wp generation with normalization starts with the invariant in core syntax and uses the *wp* function to generate a precondition with document updates. After establishing safety, we replace updates by conditionals and multiset arithmetic simulating their effects on expressions. We then remove the conditionals and in a final step eliminate the multiset arithmetic. This brings us back to a core syntax assertion such that new parts are separated from the invariant.

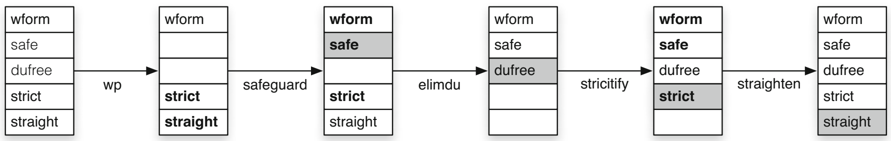
We first define conditionals and arithmetic operators on multisets:

$$\begin{array}{rl} \text{for } V, P, M \text{ and } R & \alpha ::= \ldots \mid \mathit{if}\ L\ \mathit{then}\ \alpha\ \mathit{else}\ \alpha\ \mathit{fi} \\ \textbf{value expr.} & V ::= \ldots \mid V_{SI} \otimes V \\ \textbf{path expr.} & P ::= \ldots \mid V_{SI} \otimes P \mid P \oplus P \mid P \ominus P \end{array}$$

Every expression type and the relations get a conditional, guarded by a single literal. The operator $n \otimes V$ replicates the elements of V n-times, i.e., the first argument has to denote a positive integer. The operators $\oplus$ and $\ominus$ denote multiset plus and minus for paths. With these additions – and the concepts we discussed earlier – we can now define all necessary properties:

$$\begin{array}{rl} \mathit{wform} & \equiv \text{well-formed} \\ \mathit{dufree} & \equiv \text{does not contain document updates} \\ \mathit{strict} & \equiv \text{does not contain conditionals} \\ \mathit{straight} & \equiv \text{does not contain additional multiset operations} \\ \mathit{safe} & \equiv \text{does never yield bottom (see Def. 6)} \end{array}$$

Except for safety, all other properties are purely syntactic and are defined using *xprop* (cf. Sect. 5). The following figure summarizes the normalization process. The details of the individual steps are described in the remainder of the section.



6.1 Establishing Safety

Safety is important for the elimination of updates and multiset arithmetic. As the wp-transformer might return an unsafe assertion, we introduce guards. This is realized by the function *safeguard* that adds appropriate conjuncts to the assertion. This guarding step works because the assertion language allows to express these guards for all its operators. Our function *safeguard* assumes that assertions are strict and straight.

Theorem 3 (Secure)

1. $strict\ C \ \wedge\ straight\ C \ \wedge\ \|C\|_E \neq \bot \implies \|safeguard\ C\|_E = \|C\|_E$
2. $strict\ C \ \wedge\ straight\ C \ \wedge\ \|C\|_E = \bot \implies \|safeguard\ C\|_E = false$
3. $strict\ C \ \wedge\ straight\ C \implies safe\ (safeguard\ C)$

6.2 Elimination of Document Updates

The assertion language was designed in such way that document updates can only occur as parameters of read operations. Thus, we are concerned with expressions like $V_e \equiv M[P_u \mapsto V_?](P_r)$ where P_u is the singleton path at which the update occurs and P_r is a multiset of paths at which the document is read. To make the elimination work, we need to exploit the context of V_e and the specific properties of the assertion language. We distinguish the following two cases.

Unique Path. If V_e does not appear in an expression context with a surrounding aggregate function, then we can prove that P_r denotes a singleton path, i.e., the read access is unique (using the safety and well-formedness property). Thus, we can use a case distinction to express the update:

$$\begin{aligned} elim_U\ M[P_u \mapsto V](P_r) &= if\ P_r = P_u\ then\ V\ else\ M(P_r)\ fi \\ elim_U\ M[P_u \mapsto](P_r) &= if\ P_r \in intersect\ P_r\ P_u\ then\ \{\}\ else\ M(P_r)\ fi \\ elim_U\ V &= V \end{aligned}$$

where $intersect\ P_r\ P_u$ is a syntactic function that calculates an expression representing the multiset of paths that are contained in P_r and are descendants of P_u.

Updates in Aggregates. Now, we consider document updates that have a context with a surrounding aggregate function. In such a context, P_r denotes a general multiset of paths. The basic idea is to split P_r into the multiset $P_r \ominus (count\ P_r\ P_u \otimes P_u)$ that is not affected by the update and the multiset of paths $count\ P_r\ P_u \otimes P_u$ that is affected. For the first set, we return the old values, for the other set we return new value. And similarly, for the delete operation:

$$\begin{aligned} elim_B\ M[P_u \mapsto V](P_r) &= M(P_r \ominus (count\ P_r\ P_u \otimes P_u)) \oplus (count\ P_r\ P_u \otimes V) \\ elim_B\ M[P_u \mapsto](P_r) &= M(P_r \ominus intersect\ P_r\ P_u) \\ elim_B\ V &= V \end{aligned}$$

Correctness Properties. For the definitions of $elim_U$ and $elim_B$ we proved the following properties:

Theorem 4 (Basic Eliminations Semantics)

1. $\mathit{wform}\ V \;\wedge\; \|\, V\,\|_E = \{x\} \;\wedge\; V \textit{ statically unique} \quad\Longrightarrow\quad \|\,\mathit{elim}_U\ V\,\|_E = \|\,V\,\|_E$
2. $\mathit{wform}\ V \;\wedge\; \|\, V\,\|_E \neq \bot \quad\Longrightarrow\quad \|\,\mathit{elim}_B\ V\,\|_E = \|\,V\,\|_E$

Using elim_U and elim_B , we develop a function *elimdu* that eliminates all occurrences of updates in an assertion. *elimdu* is based on the *xmap* facility in a non-trivial way. Putting this machinery together, we can prove the following central elimination properties:

Theorem 5 (Elim DM)

1. $\mathit{wform}\ C \;\wedge\; \mathit{safe}\ C \quad\Longrightarrow\quad \|\,\mathit{elimdu}\ C\,\|_E = \|\,C\,\|_E$
2. $\mathit{strict}\ C \quad\Longrightarrow\quad \mathit{dufree}\,(\mathit{elimdu}\ C)$

The proof of the semantic equivalence is based on properties of the *xmap* facility stated in Thm. 2. Besides eliminating updates, the function *elimdu* preserves well-formedness and safety.

6.3 Eliminating Conditionals

Conditionals in expressions are eliminated by pulling them up to the assertion level and replace them by corresponding logical operators. This is realized by the function *strictify* which is based on a polymorphic pull function and the *xmap* facility. The central properties are:

Theorem 6 (Strictify)

$$\|\,\mathit{strictify}\ C\,\|_E = \|\,C\,\|_E \quad \textit{and} \quad \mathit{strict}\,(\mathit{strictify}\ C)$$

Whereas this elimination is simple in principle, the direct realization in Isabelle/HOL did not work, because it produced a huge combinatorial blow up in internally generated function equations (resulting from the patterns in the function definitions). Our solution was to make the pull operation polymorphic in a type class and realize instances for each abstract syntax type. Although well separated from all other aspects, the conditional elimination constitutes an Isabelle/HOL theory of 600 lines.

6.4 Eliminating Multiset Arithmetic

We are now ready for the final normalization step, which brings us back to the core syntax by eliminating the remaining multiset arithmetic. This elimination is quite complex and not only depends on the concept of safety and well-formed assertions, as was the case in the elimination of updates, but also on the choice of the core syntax operators.

We eliminate the remaining multiset arithmetic by pulling $\otimes$, $\oplus$ and $\ominus$ out of both value and path expressions, up to an enclosing aggregate function, which we know exists because of well-formedness. The *homomorphic* property of each aggregate then allows us to pull them out of the aggregate, i.e., we can express their effect on the aggregate by using only the core operators $\oplus$ and $\odot$ on value expressions. For the pull-out process to work, we also designed the core language in such a way that all other operators which can contain multiset arithmetic are *homomorphic* too.

All these homomorphic properties are captured by a non-trivial family of functions, which consists of one function per aggregate, one for kind steps and two for path steps, as path steps can contain both path and value expressions with offending operators. As an example we show a selection of homomorphic properties, which demonstrate a lot of different cases:

$$\begin{array}{rclrcl} sum\,(V_1 \odot V_2) & \equiv & V_1 \odot sum\; V_2 & P/l[V_1 \oplus V_2] & \equiv & P/l[V_1] \oplus P/l[V_2] \\ sum\,(V_1 \otimes V_2) & \equiv & V_1 \odot sum\; V_2 & sum\; d(P_1 \ominus P_2) & \equiv & sum\; d(P_1) - sum\; d(P_2) \end{array}$$

$$count\,(V_1 \odot (V_2 \oplus V_3))\; V_4 \equiv count\,(V_1 \odot V_2)\; V_4 + count\,(V_1 \odot V_3)\; V_4$$

It starts with the simple case of the distribution law of multiplication over summation. Next, the path operator is homomorphic with regard to the multiset plus on values and paths. The third example shows how operators can change when pulled out, in this case the replication becomes a scalar multiplication. This example also shows, that all such equivalences only need to hold within the domain of the left side, as this is enough to use Thm. 2 to exploit safety.

The summation of a read operation containing a multiset minus shows that we sometimes need combinations of operators to be able to pull them out. In this case we simply don't have the minus on values, although the read operation itself is homomorphic with regard to it and the minus on paths. For this reason we also do not define a function for read, but distribute the read cases over the other aggregates. Last but not least, the count operation misses a usable homomorphic property regarding the multiplication, such that we have to pull offending operators out of both.

Based on the family of functions, we define the function *straighten* based on the *xmap* facility in a non-trivial way. We haven proven the following central elimination properties:

Theorem 7 (Normalize)

1. $safe\; C \implies \|\, straighten\; C \,\|_E = \|\, C \,\|_E$
2. $wform\; C \;\wedge\; strict\; C \implies straight\,(straighten\; C)$

As for *elimdu*, the proof of the semantic equivalence is based on safety. The well-formed property comes into play to guarantee that all operations can be eliminated. The function *straighten* of course also preserves all other properties established in the steps before.

6.5 Splitting the Generated Precondition

In Sect. 4 we remarked that the generated precondition strongly resembles the invariant and is only augmented with additional conjuncts and stacked updates at each \$. Making the result safe does not change the precondition at all, but only augments more conjuncts. The real transformation starts with *elimdu*, which removes the stacks of updates and replaces it with either conditionals or multiset arithmetic.

Conditionals are then eliminated using *strictify*, which leads to two versions of the surrounding disjunction, one which does not contain the read operation at all, but a value expression or empty set instead, and one which has exactly the form of the invariant, with added literals that only make it weaker. By this process, we gradually split up specific, much simpler versions of the original disjunction from the one staying generic – but getting weaker – which is still implied by the invariant and can be dropped at the end.

The part of the elimination based on $elim_B$ is more complicated, but looking at its two cases we notice that the original document M appears in a read operation with its original parameter P_r and some other path which is subtracted. As the *straighten* function uses homomorphisms to drag all multiset operations up to a surrounding aggregate, the original read $M(P_r)$ from the invariant is quickly recombined, split up from the other introduced expressions and its surrounding expression restored. It then depends on the context of the aggregate how the simplification can use this property. If the aggregate was embedded in an equality, for example, we can use the equality of the invariant to completely replace the aggregate in the precondition.

7 Related Work

The presented work is related to formalizations of wp-transformations and logics in interactive theorem provers, languages for data properties and schemas, and to literature about the modeling of semistructured data.

Formalizations. Weakest precondition generation has a well-established theory (see, e.g., [12]) and many researchers formalized the wp-generation in theorem provers. For example in [20], the proof assistant Coq has been used to formalize the efficient weakest precondition generation, as well as the required static single assignment and passification transformations, and prove all of them correct. In [19], Schirmer formalizes a generic wp-calculus for imperative languages in Isabelle/HOL. He leaves the state space polymorphic, whereas our contribution focuses on its design and the design and deep embedding of a matching assertion language. Related work is also concerned with formalizing the languages of the XML stack (e.g., Mini-XQuery [9]) and the soundness and correctness proofs for programming logics (e.g., [1, 17]).

Data Properties and Schemas. A lot of research on maintaining schema constraints has been done using regular languages and automata [2, 18, 3] or using complex modal logics, e.g., context logic [8, 11]. Both can handle structure more naturally and are more powerful in this regard compared to our approach. They can also handle references, but only global ones, as they lack the means to specify targets for constraints. It is therefore not surprising that it is shown in [7], that context logic formulas define regular languages. Incremental checks with these kinds of constraints are well researched, especially for automata.

The main difference to our work is our support for value-based constraints, including aggregates, even in combination with non-global references. To express these kinds of *context-dependent* constraints, we use a classical three-valued logic together with paths and a core set of aggregate functions and predicates, which allows us to combine type and integrity constraints. Precondition generation is also used for context logic in [11], but they do not support update elimination and incremental checks.

Data Models. Our work is based on a concept of paths with integrated local key constraints. With this design decision, we follow many of the ideas of Buneman et al. [4, 5, 6]. They argue for the importance of keys to uniquely identify elements, that keys should not be globally unique and that keys should be composed to "hierarchical keys" for hierarchical structures (see [5]). Their hierarchical keys resemble our core concept of paths. They advocate the usage of paths to specify key constraints and the need for a simpler language than XPath to reason about them, which in particular means paths should only move down the tree and should not contain predicates.

Although they are not in favor of document order – and argue against predicates which could observe it in paths – they use the position of elements to create the unique paths to elements they need for reasoning. By incorporating a simple variant of local key constraints in our paths, we can use element names, rather than positions, to uniquely identify elements. We believe that most of the more complex keys they suggest can be expressed in our logic. This decision, to uniformly handle typical type constraints and more complex integrity constraints within one formalism, is also discussed and backed up by [4].

8 Conclusion

Maintaining data invariants is important in many areas of computing. An efficient approach is to check invariants incrementally, i.e., make sure that invariants are established on data object creation and maintained when data objects are manipulated. To apply this approach one needs to compute the preconditions for all basic procedures manipulating the data. We are interested in automatic and efficient techniques for maintaining integrity constraints going beyond structural properties. To generate efficiently checkable preconditions for such integrity constraints, we needed a normalization process that eliminates update operations from the preconditions and splits the generated precondition into the invariant and the part to be checked.

First attempts to develop such a process by hand failed. The combination of (a) multisets resulting from kind steps in the integrity constraints, (b) intermediate syntax extensions, and (c) exceptional situation caused, e.g., by reading at paths not in the document, is very difficult to manage without tool support. The use of Isabelle/HOL was indispensable for developing succinct definitions, proving the correctess of the transformation, and generating the implementation. The resulting theory[3] consists of more than 9000 lines, the generated Haskell code for the wp-transformer about 2200. This includes the implementation and correctness proof of a first version of a simplifier for preconditions we developed.

Our next steps are the improvement of the simplifier and the development of an appropriate schema language on top of the core assertion language (similar to the one discussed in Sect. 2). With the first version of such a schema language and our approach, we made very positive experiences during the development of the student registration system [13]. This system is in practical use and very stable.

[3] The Isabelle/HOL theory files and PDFs can be found at `https://xcend.de/theory/`

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] Barbosa, D., et al.: Efficient incremental validation of XML documents. In: ICDE 2004, pp. 671–682. IEEE Computer Society, Washington, DC, USA (2004)
[3] Bouchou, B., Alves, M.H.F.: Updates and Incremental Validation of XML Documents. In: Lausen, G., Suciu, D. (eds.) DBPL 2003. LNCS, vol. 2921, pp. 216–232. Springer, Heidelberg (2004)
[4] Buneman, P., et al.: Constraints for semistructured data and XML. SIGMOD Rec. 30, 47–54 (2001)
[5] Buneman, P., et al.: Keys for XML. In: WWW 2001, pp. 201–210. ACM, Hong Kong (2001)
[6] Buneman, P., Davidson, S.B., Fan, W., Hara, C., Tan, W.C.: Reasoning about Keys for XML. In: Ghelli, G., Grahne, G. (eds.) DBPL 2001. LNCS, vol. 2397, pp. 133–148. Springer, Heidelberg (2002)
[7] Calcagno, C., Dinsdale-Young, T.: Decidability of context logic (2009)
[8] Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. In: POPL 2005, pp. 271–282. ACM, Long Beach (2005)
[9] Cheney, J., Urban, C.: Mechanizing the Metatheory of mini-XQuery. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 280–295. Springer, Heidelberg (2011)
[10] Clark, J.: RELAX NG compact syntax (November 2002), http://www.oasis-open.org/committees/relax-ng/compact-20021121.html
[11] Gardner, P.A., et al.: Local hoare reasoning about dom. In: PODS 2008, pp. 261–270. ACM, Vancouver (2008)
[12] Gries, D.: The Science of Programming. Springer (1981)
[13] Michel, P., Fillibeck, C.: Stats - softech achievement tracking system (2011), http://xcend.de/stats/start
[14] Michel, P., Poetzsch-Heffter, A.: Assertion support for manipulating constrained data-centric XML. In: PLAN-X 2009 (January 2009)
[15] Michel, P., Poetzsch-Heffter, A.: Maintaining XML Data Integrity in Programs - An Abstract Datatype Approach. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901, pp. 600–611. Springer, Heidelberg (2010)
[16] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
[17] von Oheimb, D.: Hoare logic for java in Isabelle/HOL. Concurrency and Computation: Practice and Experience 13(13), 1173–1214 (2001)
[18] Papakonstantinou, Y., Vianu, V.: Incremental Validation of XML Documents. In: Calvanese, D., Lenzerini, M., Motwani, R. (eds.) ICDT 2003. LNCS, vol. 2572, pp. 47–63. Springer, Heidelberg (2002)
[19] Schirmer, N.: A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 398–414. Springer, Heidelberg (2005)
[20] Vogels, F., Jacobs, B., Piessens, F.: A machine-checked soundness proof for an efficient verification condition generator. In: Symposium on Applied Computing, vol. 3, pp. 2517–2522. ACM (March 2010)

Bag Equivalence via a Proof-Relevant Membership Relation

Nils Anders Danielsson

Chalmers University of Technology and University of Gothenburg

Abstract. Two lists are *bag equivalent* if they are permutations of each other, i.e. if they contain the same elements, with the same multiplicity, but perhaps not in the same order. This paper describes how one can define bag equivalence as the presence of bijections between sets of membership proofs. This definition has some desirable properties:

– Many bag equivalences can be proved using a flexible form of equational reasoning.
– The definition generalises easily to arbitrary unary containers, including types with infinite values, such as streams.
– By using a slight variation of the definition one gets set equivalence instead, i.e. equality up to order and multiplicity. Other variations give the subset and subbag preorders.
– The definition works well in mechanised proofs.

1 Introduction

Bag (or *multiset*) *equivalence* is equality up to reordering of elements. For simplicity we can start by considering lists. The lists $[1, 2, 1]$ and $[2, 1, 1]$ are bag equivalent: $[1, 2, 1] \approx_{bag} [2, 1, 1]$. These lists are not bag equivalent to $[1, 2]$, because of differing multiplicities. *Set equivalence*, equality up to reordering and multiplicity, identifies all three lists: $[1, 2, 1] \approx_{set} [2, 1, 1] \approx_{set} [1, 2]$.

Bag equivalence is useful when specifying the correctness of certain algorithms. The most obvious example may be provided by sorting. The result of sorting something should be bag equivalent to the input: $\forall\, xs.\ sort\ xs \ \approx_{bag}\ xs$. In many cases the two sides of a bag equivalence (in this case *sort xs* and *xs*) have the same type, but this is not necessary. Consider tree sort, for instance:

$$\begin{aligned} &\textit{tree-sort}\ :\ \textit{List}\ \mathbb{N} \to \textit{List}\ \mathbb{N} \\ &\textit{tree-sort}\ =\ \textit{flatten} \circ \textit{to-search-tree} \end{aligned}$$

The function *to-search-tree* constructs binary search trees from lists, and *flatten* flattens trees. We can prove $\forall\, xs.\ \textit{tree-sort}\ xs \ \approx_{bag}\ xs$ by first establishing the following two lemmas:

$$\forall\, xs.\ \textit{to-search-tree}\ xs \ \approx_{bag}\ xs \qquad\qquad \forall\, t.\ \textit{flatten}\ t \ \approx_{bag}\ t$$

These lemmas relate trees and lists.

Another example of the utility of bag equivalence is provided by grammars. Two grammars are typically said to be equivalent if they generate the same language, i.e. the same *set* of strings. However, this is a coarse form of equivalence which identifies ambiguous and unambiguous grammars. If the languages are instead seen as *bags*, then one gets a form of equivalence which takes ambiguity into account.

Assume that *Grammar* represents grammars annotated with semantic actions, and that we have a function $\mathit{parse} : \mathit{Grammar} \to (\mathit{String} \to \mathit{List}\ \mathit{Result})$ which gives the semantics of a grammar as a function from strings to lists of results (multiple results in the case of ambiguous grammars). It is then reasonable to require that a program *opt* which transforms grammars into more optimised forms should satisfy the following property:

$$\forall\ g\ s.\ \mathit{parse}\ (\mathit{opt}\ g)\ s\ \approx_{set}\ \mathit{parse}\ g\ s \quad \wedge \quad \mathit{parse}\ (\mathit{opt}\ g)\ s\ \lesssim_{bag}\ \mathit{parse}\ g\ s$$

Here $\_\lesssim_{bag}\_$ is the *subbag preorder*: $xs \lesssim_{bag} ys$ if every element in xs occurs at least as often in ys. The property states that the new grammar should yield the same results as the old grammar ($\_\approx_{set}\_$), with no more ambiguity ($\_\lesssim_{bag}\_$). The order of the results is unspecified. Note that if we have infinitely ambiguous grammars, then the lists returned by *parse* can be infinite, in which case we need notions of set equivalence and subbag preorder adapted to such lists.

Many definitions of bag equivalence and related concepts are available in the literature, including classical definitions of permutations; definitions of bag equivalence for lists in the Coq [19], Ssreflect [7] and Coccinelle [5] libraries; and definitions of the type of bags in the Boom hierarchy [14], in terms of quotient containers [2], and in terms of combinatorial species [21, 13]. However, I want to propose another definition, based on bijections between sets of membership proofs (Sect. 3). This definition has several useful properties:

– It makes it possible to prove many equivalences using a flexible form of equational reasoning. This is demonstrated using examples in Sects. 4, 5 and 7.
– By modifying the definition slightly one gets definitions of set equivalence and the subset and subbag preorders (Sect. 8). By taking advantage of the similarity of these definitions one can avoid proof duplication: many preservation results, such as the fact that the list monad's bind operation preserves the various equivalences and preorders, can be established uniformly for all the relations with a single proof.
– The definition works for any type with a suitable membership predicate. Hoogendijk and de Moor [10] characterise a container type as a "relator" with an associated membership relation, so one might expect that the definition should work for many container types. Section 6 shows that it works for arbitrary unary containers, defined in the style of Abbott et al. [1]; this includes containers with infinite values, such as infinite streams.
– The definition works well in mechanised proofs, and has been used in practice: I used it to state and formally prove many properties of a parser combinator library [6].

Section 9 compares the definition to other definitions of bag equivalence.

To demonstrate that the definition works well in a formal setting I will use the dependently typed, functional language Agda [16, 18] below. The language is introduced briefly in Sect. 2. Code which includes all the main results in the text is, at the time of writing, available to download from my web page. (The code does not match the paper exactly. The main difference is that many definitions are universe-polymorphic, and hence a bit more general.)

2 Brief Introduction to Agda

In Agda one can define the types of *finite* (inductive) lists and unary natural numbers as follows:

```
data List (A : Set) : Set where        data ℕ : Set where
  []   : List A                          zero : ℕ
  _::_ : A → List A → List A             suc  : ℕ → ℕ
```

Here *Set* is a type of (small) types, and _::_ is an infix constructor; the underscores mark the argument positions. Values inhabiting inductive types can be destructed using structural recursion. For instance, the length of a list can be defined as follows:

```
length : {A : Set} → List A → ℕ
length []        = zero
length (x :: xs) = suc (length xs)
```

Here {*A* : *Set*} is an *implicit* argument. If Agda can infer such an argument uniquely from the context, then the argument does not need to be given explicitly, as witnessed by the recursive call to *length*. In some cases *explicit* arguments can be inferred from the context, and then one has the option of writing an underscore (_) instead of the full expression.

Types do not have to be defined using **data** declarations, they can also be defined using functions. For instance, we can define the type *Fin n*, which has exactly *n* elements, as follows:

```
Fin : ℕ → Set                    data _+_ (A B : Set) : Set where
Fin zero    = ⊥                    left  : A → A + B
Fin (suc n) = ⊤ + Fin n            right : B → A + B
```

Here ⊥ is the empty type, ⊤ the unit type (with sole inhabitant tt), and $A + B$ is the disjoint sum of the types A and B. By treating *Fin n* as a bounded number type we can define a safe lookup function:

```
lookup : {A : Set} (xs : List A) → Fin (length xs) → A
lookup []        ()
lookup (x :: xs) (left _)  = x
lookup (x :: xs) (right i) = lookup xs i
```

This function has a *dependent* type: the type of the index depends on the length of the list. The first clause contains an *absurd pattern*, (). This pattern is used to indicate to Agda that there are no values of type *Fin* (*length* []) = *Fin* zero = ⊥; note that type-checking can involve normalisation of terms, and that Agda would not have accepted this definition if we had omitted one of the cases.

Below we will use equivalences and bijections. One can introduce a type of equivalences between the types A and B using a record type as follows:

```
record _⇔_ (A B : Set) : Set where
  field to   : A → B
        from : B → A
```

To get a type of bijections we can add the requirement that the functions *to* and *from* are inverses:

```
record _↔_ (A B : Set) : Set where
  field to      : A → B
        from    : B → A
        from-to : ∀ x → from (to x) ≡ x
        to-from : ∀ x → to (from x) ≡ x
```

Here ∀ x → ... means the same as (x : _) → ...; Agda can infer the type of x automatically.

The type $x \equiv y$ is a type of *equality proofs* showing that x and y are equal:

```
_≡_ : {A : Set} → A → A → Set
```

I take _≡_ to be the ordinary identity type of intensional Martin-Löf type theory. In particular, I do not assume that the K rule [17], which implies that all proofs of type $x \equiv y$ are equal, is available.[1] (The reason for this choice is discussed in Sect. 10.) However, for the most part it should be fine to assume that _≡_ is the usual notion of equality used in informal mathematics.

Note that _↔_ is a *dependent record type*; later fields mention earlier ones. We can use a dependent record type to define an existential quantifier (a Σ-type):

```
record ∃ {A : Set} (B : A → Set) : Set where
  constructor _,_
  field fst : A
        snd : B fst
```

A value of type ∃ (λ (x : A) → B x) is a pair (x, y) containing a value x of type A and a value y of type B x. We can project from a record using the notation "record_type.field". For instance, ∃ comes with the following two projections:

```
∃.fst : {A : Set} {B : A → Set} → ∃ B → A
∃.snd : {A : Set} {B : A → Set} (p : ∃ B) → B (∃.fst p)
```

We can also use the existential quantifier to define the cartesian product of two types:

[1] By default the K rule is available in Agda, but in recent versions there is a flag that appears to turn it off.

$$\begin{aligned}
&\_\!\times\!\_ \;:\; \mathit{Set} \to \mathit{Set} \to \mathit{Set}\\
&A \times B \;=\; \exists\,(\lambda\,(\_ : A) \to B)
\end{aligned}$$

The relations $\_\!\Leftrightarrow\!\_$ and $\_\!\leftrightarrow\!\_$ are equivalence relations. We can for instance prove that $\_\!\leftrightarrow\!\_$ is symmetric in the following way:

$$\begin{aligned}
&\mathit{sym} \;:\; \{A\;B \;:\; \mathit{Set}\} \;\to\; A \leftrightarrow B \;\to\; B \leftrightarrow A\\
&\mathit{sym}\;p \;=\; \mathbf{record}\;\{\,\mathit{to} \;=\; \_\!\leftrightarrow\!\_.\mathit{from}\;p\;;\;\mathit{from\text{-}to} \;=\; \_\!\leftrightarrow\!\_.\mathit{to\text{-}from}\;p\\
&\qquad\qquad\qquad\;;\;\mathit{from} \;=\; \_\!\leftrightarrow\!\_.\mathit{to}\;\;p\;;\;\mathit{to\text{-}from} \;=\; \_\!\leftrightarrow\!\_.\mathit{from\text{-}to}\;p\}
\end{aligned}$$

I will also use the following combinators, corresponding to reflexivity and transitivity:

$$\begin{aligned}
&\_\Box \qquad\quad\;:\; (A \;:\; \mathit{Set}) \;\to\; A \leftrightarrow A\\
&\_\!\leftrightarrow\!\langle\_\rangle\_ \;:\; (A \;:\; \mathit{Set})\;\{B\;C \;:\; \mathit{Set}\} \;\to\\
&\qquad\qquad\quad A \leftrightarrow B \;\to\; B \leftrightarrow C \;\to\; A \leftrightarrow C
\end{aligned}$$

Here $\_\Box$ is a unary postfix operator and $\_\!\leftrightarrow\!\langle\_\rangle\_$ a right-associative ternary mixfix operator. The choice of names and the choice of which arguments are explicit and which are implicit may appear strange, but they allow us to use a notation akin to equational reasoning for "bijectional reasoning". For instance, if we have proofs $p : A \leftrightarrow B$ and $q : C \leftrightarrow B$, then we can prove $A \leftrightarrow C$ as follows:

$$\begin{aligned}
&A \quad \leftrightarrow\langle\; p \;\rangle\\
&B \quad \leftrightarrow\langle\; \mathit{sym}\;q \;\rangle\\
&C \quad \Box
\end{aligned}$$

The idea to use mixfix operators to mimic equational reasoning notation comes from Norell [16].

To avoid clutter I will usually suppress implicit argument declarations below.

3 Bag Equivalence for Lists

For simplicity, let us start by restricting the discussion to (finite) lists. When are two lists xs and ys bag equivalent? One answer: when there is a bijection f from the positions of xs to the positions of ys, such that the value at position i in xs is equal to the value at position $f\;i$ in ys. We can formalise this as follows:

$$\begin{aligned}
&\mathbf{record}\;\_\!\approx'_{\mathit{bag}}\!\_\;(\mathit{xs}\;\mathit{ys} \;:\; \mathit{List}\;A) \;:\; \mathit{Set}\;\mathbf{where}\\
&\quad\mathbf{field}\;\mathit{bijection} \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs}) \leftrightarrow \mathit{Fin}\;(\mathit{length}\;\mathit{ys})\\
&\qquad\qquad\;\mathit{related} \quad\;:\; \forall\; i \to \mathit{lookup}\;\mathit{xs}\;i \equiv \mathit{lookup}\;\mathit{ys}\;(\_\!\leftrightarrow\!\_.\mathit{to}\;\mathit{bijection}\;i)
\end{aligned}$$

However, I prefer a different (but equivalent) definition.

Let us first define the Any predicate transformer [15]:

$$\begin{aligned}
&\mathit{Any} \;:\; (A \to \mathit{Set}) \to \mathit{List}\;A \to \mathit{Set}\\
&\mathit{Any}\;P\;[\,] \qquad\;\; = \;\bot\\
&\mathit{Any}\;P\;(x :: \mathit{xs}) \;=\; P\;x + \mathit{Any}\;P\;\mathit{xs}
\end{aligned}$$

$Any\ P\ xs$ holds if $P\ x$ holds for at least one element x of xs: $Any\ P\ [x_1, \ldots, x_n]$ reduces to $P\ x_1 + \ldots + P\ x_n + \bot$. Using Any we can define a list membership predicate:

$$\begin{aligned} &\_{\in}\_ \ :\ A \to List\ A \to Set \\ &x \in xs\ =\ Any\ (\lambda\ y \to x \equiv y)\ xs \end{aligned}$$

This can be read as "x is a member of xs if there is any element y of xs which is equal to x": $x \in [x_1, \ldots, x_n]\ =\ (x \equiv x_1)\ +\ \ldots\ +\ (x \equiv x_n)\ +\ \bot$. Note that $x \in xs$ is basically a subset of the positions of xs, namely those positions which contain x. Bag equivalence can then be defined as follows:

$$\begin{aligned} &\_{\approx_{bag}}\_ \ :\ List\ A \to List\ A \to Set \\ &xs\ \approx_{bag}\ ys\ =\ \forall\ z\ \to\ z \in xs\ \leftrightarrow\ z \in ys \end{aligned}$$

Two lists xs and ys are bag equivalent if, for any element z, the type of positions $z \in xs$ is isomorphic to (in bijective correspondence with) $z \in ys$.

It is important that $x \in xs$ can (in general) contain more than one value, i.e. that the relation is "proof-relevant". This explains the title of the paper: bag equivalence via a *proof-relevant* membership relation. If the relation were proof-*irrelevant*, i.e. if any two elements of $x \in xs$ were identified, then we would get set equivalence instead of bag equivalence.

The intuitive explanation above has a flaw. It is based on the unstated assumption that the equality type itself is proof-irrelevant: if there are several distinct proofs of $x \equiv x$, then $x \in [x]$ does *not* correspond directly to the positions of x in $[x]$. However, in the absence of the K rule the equality type is *not* necessarily proof-irrelevant [9]. Fortunately, and maybe surprisingly, one can prove that the two definitions of bag equivalence above are equivalent even in the absence of proof-irrelevance (see Sect. 5).

The first definition of bag equivalence above is, in some sense, less complicated than $\_{\approx_{bag}}\_$, because it does not in general involve equality of equality proofs. One may hence wonder what the point of the new, less intuitive, more complicated definition is. My main answer to this question is that $\_{\approx_{bag}}\_$ lends itself well to bijectional reasoning.

4 Bijectional Reasoning

How can we prove that two lists are bag equivalent? In this section I will use an example to illustrate some of the techniques that are available. The task is the following: prove that bind distributes from the left over append,

$$xs \gg\!= (\lambda\ x \to f\ x +\!\!+\ g\ x)\ \approx_{bag}\ (xs \gg\!= f) +\!\!+ (xs \gg\!= g).$$

Here bind is defined as follows:

$$\begin{aligned} &\_{\gg\!=}\_ \ :\ List\ A \to (A \to List\ B) \to List\ B \\ &xs \gg\!= f\ =\ concat\ (map\ f\ xs) \end{aligned}$$

The *concat* function flattens a list of lists, *map* applies a function to every element in a list, and $\_{+\!\!+}\_$ appends one list to another.

Bag equivalence is reflexive, so any equation which holds for ordinary list equality also holds for bag equivalence. To see that the equation above does not (in general) hold for ordinary list equality, let *xs* be $1 :: 2 :: []$ and f and g both be $\lambda\ x \to x :: []$, in which case the equivalence specialises as follows: $1 :: 1 :: 2 :: 2 :: [] \approx_{bag} 1 :: 2 :: 1 :: 2 :: []$.

Before proving the left distributivity law I will introduce some basic lemmas. The first one states that *Any* is homomorphic with respect to $\_{+\!\!+}\_/\_{+}\_$. The lemma is proved by induction on the structure of the first list:

```
Any-++ : (P : A → Set) (xs ys : List A) →
            Any P (xs ++ ys) ↔ Any P xs + Any P ys
Any-++ P [] ys =
  Any P ys          ↔⟨ sym +-left-identity ⟩
  ⊥ + Any P ys   □
Any-++ P (x :: xs) ys =
  P x + Any P (xs ++ ys)           ↔⟨ +-cong (P x □) (Any-++ P xs ys) ⟩
  P x + (Any P xs + Any P ys)   ↔⟨ +-assoc ⟩
  (P x + Any P xs) + Any P ys   □
```

Note that the list *xs* in the recursive call *Any-++ P xs ys* is structurally smaller than the input, $x :: xs$. The proof uses the following lemmas:

```
+-left-identity : ⊥ + A ↔ A
+-assoc         : A + (B + C) ↔ (A + B) + C
+-cong          : A₁ ↔ A₂ → B₁ ↔ B₂ →
                  A₁ + B₁ ↔ A₂ + B₂
```

They state that the empty type is a left identity of $\_{+}\_$, and that $\_{+}\_$ is associative and preserves bijections. These lemmas can all be proved by defining two simple functions and proving that they are inverses.

Some readers may wonder why I did not include the step $Any\ P\ ([]\ {+\!\!+}\ ys) \leftrightarrow Any\ P\ ys$ in the first case of *Any-++*. This step can be omitted because the two sides are equal *by definition*: $[]\ {+\!\!+}\ ys$ reduces to *ys*. For the same reason the step $Any\ P\ ((x :: xs)\ {+\!\!+}\ ys) \leftrightarrow P\ x + Any\ P\ (xs\ {+\!\!+}\ ys)$, which involves two reductions, can be omitted in the lemma's second case.

Note that if *Any-++* is applied to $\_{\equiv}\_\ z$, then we get that list membership is homomorphic with respect to $\_{+\!\!+}\_/\_{+}\_$: $z \in xs\ {+\!\!+}\ ys \leftrightarrow z \in xs + z \in ys$. We can use this fact to prove that $\_{+\!\!+}\_$ is commutative:

```
++-comm : (xs ys : List A) → xs ++ ys ≈bag ys ++ xs
++-comm xs ys = λ z →
  z ∈ xs ++ ys          ↔⟨ Any-++ (_≡_ z) xs ys ⟩
  z ∈ xs + z ∈ ys     ↔⟨ +-comm ⟩
  z ∈ ys + z ∈ xs     ↔⟨ sym (Any-++ (_≡_ z) ys xs) ⟩
  z ∈ ys ++ xs          □
```

$$\begin{array}{ll}
x \equiv y \;\rightarrow\; (z \equiv x) \;\leftrightarrow\; (z \equiv y) & A \times \bot \;\leftrightarrow\; \bot \\
\forall\, x \;\rightarrow\; (\exists\, \lambda\, y \rightarrow y \equiv x) \;\leftrightarrow\; \top & A \times \top \;\leftrightarrow\; A \\
B\; x \;\leftrightarrow\; (\exists\, \lambda\, y \rightarrow B\; y \times y \equiv x) & (A + B) \times C \;\leftrightarrow\; (A \times C) + (B \times C) \\
 & (\exists\, \lambda\, x \rightarrow B\; x + C\; x) \;\leftrightarrow\; \exists\, B + \exists\, C \\
 & \\
(\exists\, \lambda\, (i : \mathit{Fin}\; \mathsf{zero}) \rightarrow P\; i) \;\leftrightarrow\; \bot & (\exists\, \lambda\, (i : \mathit{Fin}\; (\mathsf{suc}\; n)) \rightarrow P\; i) \;\leftrightarrow \\
 & P\; (\mathsf{left}\; \mathsf{tt}) + (\exists\, \lambda\, (i : \mathit{Fin}\; n) \rightarrow P\; (\mathsf{right}\; i)) \\
 & \\
A_1 \;\leftrightarrow\; A_2 \;\rightarrow\; B_1 \;\leftrightarrow\; B_2 \;\rightarrow & (\exists\, \lambda\, (x : A) \rightarrow \exists\, \lambda\, (y : B) \rightarrow C\; x\; y) \;\leftrightarrow \\
A_1 \times B_1 \;\leftrightarrow\; A_2 \times B_2 & (\exists\, \lambda\, (y : B) \rightarrow \exists\, \lambda\, (x : A) \rightarrow C\; x\; y)
\end{array}$$

$$(p : A_1 \;\leftrightarrow\; A_2) \;\rightarrow\; (\forall\, x \rightarrow B_1\; x \;\leftrightarrow\; B_2\; (\_\!\leftrightarrow\!\_.\mathit{to}\; p\; x)) \;\rightarrow\; \exists\, B_1 \;\leftrightarrow\; \exists\, B_2$$

Fig. 1. Unnamed lemmas used in proofs in Sects. 4–5 (some are consequences of others)

Here I have used the fact that $\_+\_$ is commutative: $+\text{-}\mathit{comm} : A + B \;\leftrightarrow\; B + A$. Note how commutativity of $\_+\!\!+\_$ follows from commutativity of $\_+\_$.

In the remainder of the text I will conserve space and reduce clutter by not writing out the explanations within brackets, such as $\langle\; +\text{-}\mathit{comm}\; \rangle$. For completeness I list various (unnamed) lemmas used in the proofs below in Fig. 1.

Let us now consider two lemmas that relate *Any* with *concat* and *map*:

$$\begin{array}{lcll}
\multicolumn{4}{l}{\mathit{Any\text{-}concat} : (P : A \rightarrow \mathit{Set})\; (\mathit{xss} : \mathit{List}\; (\mathit{List}\; A)) \rightarrow} \\
\multicolumn{4}{l}{\qquad\qquad\quad \mathit{Any}\; P\; (\mathit{concat}\; \mathit{xss}) \;\leftrightarrow\; \mathit{Any}\; (\mathit{Any}\; P)\; \mathit{xss}} \\
\mathit{Any\text{-}concat}\; P\; [\,] & = & \bot & \square \\
\mathit{Any\text{-}concat}\; P\; (\mathit{xs} :: \mathit{xss}) & = & \mathit{Any}\; P\; (\mathit{xs} +\!\!+ \mathit{concat}\; \mathit{xss}) & \leftrightarrow \\
 & & \mathit{Any}\; P\; \mathit{xs} + \mathit{Any}\; P\; (\mathit{concat}\; \mathit{xss}) & \leftrightarrow \\
 & & \mathit{Any}\; P\; \mathit{xs} + \mathit{Any}\; (\mathit{Any}\; P)\; \mathit{xss} & \square
\end{array}$$

$$\begin{array}{lcll}
\multicolumn{4}{l}{\mathit{Any\text{-}map} : (P : B \rightarrow \mathit{Set})\; (f : A \rightarrow B)\; (\mathit{xs} : \mathit{List}\; A) \rightarrow} \\
\multicolumn{4}{l}{\qquad\qquad \mathit{Any}\; P\; (\mathit{map}\; f\; \mathit{xs}) \;\leftrightarrow\; \mathit{Any}\; (P \circ f)\; \mathit{xs}} \\
\mathit{Any\text{-}map}\; P\; f\; [\,] & = & \bot & \square \\
\mathit{Any\text{-}map}\; P\; f\; (x :: \mathit{xs}) & = & P\; (f\; x) \;\;\, + \mathit{Any}\; P\; (\mathit{map}\; f\; \mathit{xs}) & \leftrightarrow \\
 & & (P \circ f)\; x + \mathit{Any}\; (P \circ f)\; \mathit{xs} & \square
\end{array}$$

Here $\_\circ\_$ is function composition. If we combine *Any-concat* and *Any-map*, then we can also relate *Any* and bind:

$$\begin{array}{lcll}
\multicolumn{4}{l}{\mathit{Any}\text{-}\!\gg\!= : (P : B \rightarrow \mathit{Set})\; (\mathit{xs} : \mathit{List}\; A)\; (f : A \rightarrow \mathit{List}\; B) \rightarrow} \\
\multicolumn{4}{l}{\qquad\qquad \mathit{Any}\; P\; (\mathit{xs} \gg\!= f) \;\leftrightarrow\; \mathit{Any}\; (\mathit{Any}\; P \circ f)\; \mathit{xs}} \\
\mathit{Any}\text{-}\!\gg\!=\; P\; \mathit{xs}\; f & = & \mathit{Any}\; P\; (\mathit{concat}\; (\mathit{map}\; f\; \mathit{xs})) & \leftrightarrow \\
 & & \mathit{Any}\; (\mathit{Any}\; P)\; (\mathit{map}\; f\; \mathit{xs}) & \leftrightarrow \\
 & & \mathit{Any}\; (\mathit{Any}\; P \circ f)\; \mathit{xs} & \square
\end{array}$$

Note that these lemmas allow us to move things between the two arguments of *Any*, the list and the predicate. When defining bag equivalence I could have defined the list membership predicate $\_\in\_$ directly, without using *Any*, but I like the flexibility which *Any* provides.

Sometimes it can be useful to switch between *Any* and $\_\in\_$ using the following lemma (which can be proved by induction on *xs*):

$$\textit{Any-}{\in} \;:\; \textit{Any}\ P\ \textit{xs}\ \leftrightarrow\ (\exists\ \lambda\ x \rightarrow P\ x \times x \in \textit{xs})$$

This lemma can for instance be used to show that *Any* preserves bijections and respects bag equivalence:

$$\begin{array}{l}
\textit{Any-cong} \;:\; (P\ Q \;:\; A \rightarrow \textit{Set})\ (\textit{xs}\ \textit{ys} \;:\; \textit{List}\ A) \rightarrow \\
\qquad\qquad\quad (\forall\ x\ \rightarrow\ P\ x\ \leftrightarrow\ Q\ x)\ \rightarrow\ \textit{xs}\ \approx_{\textit{bag}}\ \textit{ys}\ \rightarrow \\
\qquad\qquad\quad \textit{Any}\ P\ \textit{xs}\ \leftrightarrow\ \textit{Any}\ Q\ \textit{ys} \\
\textit{Any-cong}\ P\ Q\ \textit{xs}\ \textit{ys}\ p\ \textit{eq}\ = \\
\quad \begin{array}{ll}
\textit{Any}\ P\ \textit{xs} & \leftrightarrow \\
(\exists\ \lambda\ z \rightarrow P\ z \times z \in \textit{xs}) & \leftrightarrow \\
(\exists\ \lambda\ z \rightarrow Q\ z \times z \in \textit{ys}) & \leftrightarrow \\
\textit{Any}\ Q\ \textit{ys} & \square
\end{array}
\end{array}$$

We can now prove the left distributivity law using the following non-recursive definition:

$$\begin{array}{l}
{\gg\!=}\textit{-left-distributive} \;:\; (\textit{xs} \;:\; \textit{List}\ A)\ (f\ g \;:\; A \rightarrow \textit{List}\ B) \rightarrow \\
\quad \textit{xs} \gg\!= (\lambda\ x \rightarrow f\ x \mathbin{+\!\!+} g\ x)\ \approx_{\textit{bag}}\ (\textit{xs} \gg\!= f) \mathbin{+\!\!+} (\textit{xs} \gg\!= g) \\
{\gg\!=}\textit{-left-distributive}\ \textit{xs}\ f\ g\ =\ \lambda\ z \rightarrow \\
\quad \begin{array}{ll}
z\ \in\ \textit{xs} \gg\!= (\lambda\ x \rightarrow f\ x \mathbin{+\!\!+} g\ x) & \leftrightarrow \\
\textit{Any}\ (\lambda\ x \rightarrow z \in f\ x \mathbin{+\!\!+} g\ x)\ \textit{xs} & \leftrightarrow \\
\textit{Any}\ (\lambda\ x \rightarrow z \in f\ x\ +\ z \in g\ x)\ \textit{xs} & \leftrightarrow \\
\textit{Any}\ (\lambda\ x \rightarrow z \in f\ x)\ \textit{xs} + \textit{Any}\ (\lambda\ x \rightarrow z \in g\ x)\ \textit{xs} & \leftrightarrow \\
z \in \textit{xs} \gg\!= f\ +\ z \in \textit{xs} \gg\!= g & \leftrightarrow \\
z\ \in\ (\textit{xs} \gg\!= f) \mathbin{+\!\!+} (\textit{xs} \gg\!= g) & \square
\end{array}
\end{array}$$

The proof amounts to starting from both sides, using the lemmas introduced above to make the list arguments as simple as possible, and finally proving the following lemma in order to tie the two sides together in the middle:

$$\begin{array}{l}
\textit{Any-+} \;:\; (P\ Q \;:\; A \rightarrow \textit{Set})\ (\textit{xs} \;:\; \textit{List}\ A) \rightarrow \\
\qquad\qquad \textit{Any}\ (\lambda\ x \rightarrow P\ x + Q\ x)\ \textit{xs}\ \leftrightarrow\ \textit{Any}\ P\ \textit{xs} + \textit{Any}\ Q\ \textit{xs} \\
\textit{Any-+}\ P\ Q\ \textit{xs}\ = \\
\quad \begin{array}{ll}
\textit{Any}\ (\lambda\ x \rightarrow P\ x + Q\ x)\ \textit{xs} & \leftrightarrow \\
(\exists\ \lambda\ x \rightarrow (P\ x + Q\ x) \times x \in \textit{xs}) & \leftrightarrow \\
(\exists\ \lambda\ x \rightarrow P\ x \times x \in \textit{xs} + Q\ x \times x \in \textit{xs}) & \leftrightarrow \\
(\exists\ \lambda\ x \rightarrow P\ x \times x \in \textit{xs}) + (\exists\ \lambda\ x \rightarrow Q\ x \times x \in \textit{xs}) & \leftrightarrow \\
\textit{Any}\ P\ \textit{xs} + \textit{Any}\ Q\ \textit{xs} & \square
\end{array}
\end{array}$$

Note how the left distributivity property for bind is reduced to the facts that $\_\times\_$ and $\exists$ distribute over $\_+\_$ (second and third steps above).

The example above suggests that the definition of bag equivalence presented in this paper makes it possible to establish equivalences in a *modular* way, using a *flexible* form of equational reasoning: even though we are establishing a correspondence of the form $\textit{xs}\ \approx_{\textit{bag}}\ \textit{ys}$ the reasoning need not have the form $\textit{xs}\ \approx_{\textit{bag}}\ \textit{xs}'\ \approx_{\textit{bag}}\ \ldots\ \approx_{\textit{bag}}\ \textit{ys}$.

5 The Definitions Are Equivalent

Before generalising the definition of bag equivalence I want to show that the two definitions given in Sect. 3 are equivalent.

Let us start by showing that $\_{\approx_{bag}}\_$ is complete with respect to $\_{\approx'_{bag}}\_$. We can relate the membership predicate and the lookup function as follows:

$$\in\text{-}\mathit{lookup} \;:\; z \in \mathit{xs} \;\leftrightarrow\; \exists\,(\lambda\,(i \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs})) \rightarrow z \equiv \mathit{lookup}\;\mathit{xs}\;i)$$

This lemma can be proved by induction on the list xs. It is then easy to establish completeness:

$$\begin{array}{l}
\mathit{complete} \;:\; (\mathit{xs}\;\mathit{ys} \;:\; \mathit{List}\;A) \;\rightarrow\; \mathit{xs} \;\approx'_{bag}\; \mathit{ys} \;\rightarrow\; \mathit{xs} \;\approx_{bag}\; \mathit{ys} \\
\mathit{complete}\;\mathit{xs}\;\mathit{ys}\;\mathit{eq} \;=\; \lambda\, z \rightarrow \\
\quad \begin{array}{ll}
z \in \mathit{xs} & \leftrightarrow \\
\exists\,(\lambda\,(i \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs})) \rightarrow z \equiv \mathit{lookup}\;\mathit{xs}\;i) & \leftrightarrow \\
\exists\,(\lambda\,(i \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{ys})) \rightarrow z \equiv \mathit{lookup}\;\mathit{ys}\;i) & \leftrightarrow \\
z \in \mathit{ys} & \square
\end{array}
\end{array}$$

The second step uses the two components of eq.

Using the $\in$-*lookup* lemma we can also construct an isomorphism between the type of positions $\exists\,\lambda\, z \rightarrow z \in \mathit{xs}$ and the corresponding type of indices:

$$\begin{array}{l}
\mathit{Fin\text{-}length} \;:\; (\mathit{xs} \;:\; \mathit{List}\;A) \;\rightarrow\; (\exists\,\lambda\, z \rightarrow z \in \mathit{xs}) \;\leftrightarrow\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs}) \\
\mathit{Fin\text{-}length}\;\mathit{xs} \;= \\
\quad \begin{array}{ll}
(\exists\,\lambda\, z \rightarrow z \in \mathit{xs}) & \leftrightarrow \\
(\exists\,\lambda\, z \rightarrow \exists\,\lambda\,(i \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs})) \rightarrow z \equiv \mathit{lookup}\;\mathit{xs}\;i) & \leftrightarrow \\
(\exists\,\lambda\,(i \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs})) \rightarrow \exists\,\lambda\, z \rightarrow z \equiv \mathit{lookup}\;\mathit{xs}\;i) & \leftrightarrow \\
\mathit{Fin}\;(\mathit{length}\;\mathit{xs}) \times \top & \leftrightarrow \\
\mathit{Fin}\;(\mathit{length}\;\mathit{xs}) & \square
\end{array}
\end{array}$$

The penultimate step uses the fact that, for any x, types of the form $\exists\,\lambda\, y \rightarrow y \equiv x$ are "contractible" [20, Lemma `idisweq`], and hence isomorphic to the unit type. One can easily reduce this fact to the problem of proving that (x, refl) is equal to (y, eq), for arbitrary y and $\mathit{eq} \,:\, y \equiv x$, where $\mathsf{refl} \,:\, \{A \,:\, \mathit{Set}\}\;\{z \,:\, A\} \rightarrow z \equiv z$ is the canonical proof of reflexivity. This follows from a single application of the J rule—the usual eliminator for the Martin-Löf identity type—which in this case allows us to pattern match on eq, replacing it with refl and unifying y and x.

As an aside one can note that *Fin-length* is a generalisation of the fact above (this observation is due to Thierry Coquand). The statement of *Fin-length* may be a bit more suggestive if the existential is written as a Σ-type:

$$(\Sigma\; x \;:\; A.\; x \equiv x_1 + \ldots + x \equiv x_n) \;\leftrightarrow\; \mathit{Fin}\;n.$$

Note that this statement is proved without assuming that the equality type is proof-irrelevant. We can for instance instantiate A with the universe *Set* and all the x_i with the type $\mathbb{N}$.[2] In homotopy type theory [20] there are infinitely many distinct proofs of $\mathbb{N} \equiv \mathbb{N}$, but *Fin-length* is still valid.

We can use *Fin-length* to construct an index bijection from a bag equivalence:

$$
\begin{array}{l}
\textit{Fin-length-cong} : (\textit{xs}\ \textit{ys} : \textit{List}\ A) \to \textit{xs} \approx_{bag} \textit{ys} \to \\
\qquad\qquad\qquad\qquad \textit{Fin}\ (\textit{length}\ \textit{xs}) \leftrightarrow \textit{Fin}\ (\textit{length}\ \textit{ys}) \\
\textit{Fin-length-cong}\ \textit{xs}\ \textit{ys}\ \textit{eq} = \\
\quad \textit{Fin}\ (\textit{length}\ \textit{xs}) \quad \leftrightarrow \\
\quad \exists\ (\lambda\ z \to z \in \textit{xs}) \quad \leftrightarrow \\
\quad \exists\ (\lambda\ z \to z \in \textit{ys}) \quad \leftrightarrow \\
\quad \textit{Fin}\ (\textit{length}\ \textit{ys}) \quad \square
\end{array}
$$

All that remains in order to establish soundness of $\_\approx_{bag}\_$ with respect to $\_\approx'_{bag}\_$ is to show that the positions which the bijection *Fin-length-cong xs ys eq* relates contain equal elements. This bijection is defined using a number of lemmas which I have postulated above. If these lemmas are instantiated with concrete definitions in a suitable way (as in the code which accompanies the paper), then the result can be established using a short proof. Thus we get soundness:

$$
\textit{sound} : (\textit{xs}\ \textit{ys} : \textit{List}\ A) \to \textit{xs} \approx_{bag} \textit{ys} \to \textit{xs} \approx'_{bag} \textit{ys}
$$

6 Bag Equivalence for Arbitrary Containers

The definition of bag equivalence given in Sect. 3 generalises from lists to many other types. Whenever we can define the *Any* type we get a corresponding notion of bag equivalence. The definition is not limited to types with finite values. We can for instance define *Any* for infinite streams (but in that case *Any* can not be defined by structural recursion as in Sect. 3).

It turns out that *containers*, in the style of Abbott et al. [1], make it very easy to define *Any*. The unary containers which I will present below can be used to represent arbitrary strictly positive simple types in one variable (in a certain extensional type theory [1]), so we get a definition of bag equivalence which works for a very large set of types. By using n-ary containers, or indexed containers [4], it should be possible to handle even more types, but I fear that the extra complexity would obscure the main idea, so I stick to unary containers here.

A (unary) container consists of a type of shapes and, for every shape, a type of positions:

$$
\begin{array}{l}
\textbf{record}\ \textit{Container} : \textit{Set}_1\ \textbf{where} \\
\quad \textbf{constructor}\ \_\rhd\_ \\
\quad \textbf{field}\ \textit{Shape} \quad : \textit{Set} \\
\qquad\qquad \textit{Position} : \textit{Shape} \to \textit{Set}
\end{array}
\qquad
\begin{array}{l}
[\![\_]\!] : \textit{Container} \to \textit{Set} \to \textit{Set} \\
[\![\ S \rhd P\ ]\!]\ A = \\
\quad \exists\ \lambda\ (s : S) \to (P\ s \to A)
\end{array}
$$

[2] After making the definitions universe-polymorphic; see the accompanying code.

(Set_1 is a type of large types.) A container C can be interpreted as a type constructor $[\![C]\!]$. Values of type $[\![S \rhd P]\!]\ A$ have the form (s, f), where s is a shape and f is a function mapping the positions corresponding to s to values.

Let us take some examples:

– We can represent finite lists using $\mathbb{N} \rhd \mathit{Fin}$: the shape is the length of the list, and a list of length n has n positions.
– Infinite streams can be represented as follows: $\top \rhd (\lambda\ \_ \to \mathbb{N})$. There is only one shape, and this shape comes with infinitely many positions.
– Consider finite binary trees with values in the internal nodes:

```
data Tree (A : Set) : Set where
  leaf : Tree A
  node : Tree A → A → Tree A → Tree A
```

This type can be represented by $S \rhd P$, where S and P are defined as follows (note that Agda supports overloaded constructors):

```
data S : Set where              P : S → Set
  leaf : S                      P leaf         = ⊥
  node : S → S → S              P (node l r)   = P l + ⊤ + P r
```

The shapes are unlabelled finite binary trees, and the positions are paths to the internal nodes.

Note that the type of shapes can be obtained by applying the container's type constructor to the unit type. For instance, S is isomorphic to $\mathit{Tree}\ \top$.

Given a container we can define Any as follows [3] (where I have written out the implicit argument $\{S \rhd P\}$ in order to be able to give a type signature for p):

```
Any : {C : Container} {A : Set} → (A → Set) → ([[ C ]] A → Set)
Any {S ▷ P} Q (s, f) = ∃ λ (p : P s) → Q (f p)
```

$\mathit{Any}\ Q\ (s, f)$ consists of pairs (p, q) where p is an s-indexed position and q is a proof showing that the value at position p satisfies the predicate Q.

We can now define bag equivalence as before. In fact, we can define bag equivalence for values of *different* container types, as long as the elements they contain have the same type:

```
_∈_ : A → [[ C ]] A → Set               _≈bag_ : [[ C1 ]] A → [[ C2 ]] A → Set
x ∈ xs = Any (λ y → x ≡ y) xs           xs ≈bag ys =
                                          ∀ z → z ∈ xs ↔ z ∈ ys
```

We can also generalise the alternative definition $\_\approx'_{bag}\_$ from Sect. 3:

```
_≈′bag_ : {C1 C2 : Container} {A : Set} → [[ C1 ]] A → [[ C2 ]] A → Set
_≈′bag_ {S1 ▷ P1} {S2 ▷ P2} (s1, f1) (s2, f2) =
  ∃ λ (b : P1 s1 ↔ P2 s2) → ∀ p → f1 p ≡ f2 (_↔_.to b p)
```

This definition states that two values are bag equivalent if there is a bijection between their positions which relates equal elements. As before $\_{\approx_{bag}}\_$ and $\_{\approx'_{bag}}\_$ are equivalent. The proof is easier than the one in Sect. 5: the generalisation of $\in$-*lookup* holds by definition.

7 More Bijectional Reasoning

Let us now revisit the tree sort example from the introduction. To avoid minor complications related to the container encoding I use the direct definition of the *Tree* type from Sect. 6, and define *Any* and membership explicitly:

$$\begin{array}{l} \mathit{Any}_{\mathit{Tree}} : (A \to \mathit{Set}) \to (\mathit{Tree}\ A \to \mathit{Set}) \\ \mathit{Any}_{\mathit{Tree}}\ P\ \mathsf{leaf} \qquad\quad = \bot \\ \mathit{Any}_{\mathit{Tree}}\ P\ (\mathsf{node}\ l\ x\ r) = \\ \quad \mathit{Any}_{\mathit{Tree}}\ P\ l + P\ x + \mathit{Any}_{\mathit{Tree}}\ P\ r \end{array}$$

$$\begin{array}{l} \_{\in_{\mathit{Tree}}}\_ : A \to \mathit{Tree}\ A \to \mathit{Set} \\ x \in_{\mathit{Tree}} t = \\ \quad \mathit{Any}_{\mathit{Tree}}\ (\lambda\ y \to x \equiv y)\ t \end{array}$$

The *flatten* function can be defined (inefficiently) as follows:

$$\begin{array}{lcl} \mathit{flatten} : \mathit{Tree}\ A \to \mathit{List}\ A \\ \mathit{flatten}\ \mathsf{leaf} & = & [\,] \\ \mathit{flatten}\ (\mathsf{node}\ l\ x\ r) & = & \mathit{flatten}\ l + \!\!+\ x :: \mathit{flatten}\ r \end{array}$$

The *flatten* lemma from the introduction can then be proved as follows (where $\_{\in_{List}}\_$ refers to the definition of list membership from Sect. 3):

$$\begin{array}{l} \mathit{flatten\text{-}lemma} : (t : \mathit{Tree}\ A) \to \forall\ z \to z \in_{\mathit{List}} \mathit{flatten}\ t \leftrightarrow z \in_{\mathit{Tree}} t \\ \mathit{flatten\text{-}lemma}\ \mathsf{leaf} \qquad\qquad = \lambda\ z \to \bot \quad \Box \\ \mathit{flatten\text{-}lemma}\ (\mathsf{node}\ l\ x\ r) = \lambda\ z \to \\ \quad \begin{array}{lcccccl} z \in_{\mathit{List}} \mathit{flatten}\ l + \!\!+\ x :: \mathit{flatten}\ r & & & & & & \leftrightarrow \\ z \in_{\mathit{List}} \mathit{flatten}\ l & + & z \equiv x & + & z \in_{\mathit{List}} \mathit{flatten}\ r & & \leftrightarrow \\ z \in_{\mathit{Tree}} l & + & z \equiv x & + & z \in_{\mathit{Tree}} r & & \Box \end{array} \end{array}$$

In the leaf case the two sides evaluate to the empty type. The node case contains two steps: the first one uses *Any*-$+\!\!+$, and the second one uses the inductive hypothesis twice.

With a suitable definition of *to-search-tree* it is not much harder to prove the following lemma (see the accompanying code):

$$\begin{array}{l} \mathit{to\text{-}search\text{-}tree\text{-}lemma} : \\ \quad (\mathit{xs} : \mathit{List}\ \mathbb{N}) \to \forall\ z \to z \in_{\mathit{Tree}} \mathit{to\text{-}search\text{-}tree}\ \mathit{xs} \leftrightarrow z \in_{\mathit{List}} \mathit{xs} \end{array}$$

It is then easy to prove that *tree-sort* produces a permutation of its input:

$$\begin{array}{l} \mathit{tree\text{-}sort\text{-}permutes} : (\mathit{xs} : \mathit{List}\ \mathbb{N}) \to \mathit{tree\text{-}sort}\ \mathit{xs} \approx_{\mathit{bag}} \mathit{xs} \\ \mathit{tree\text{-}sort\text{-}permutes}\ \mathit{xs} = \lambda\ z \to \\ \quad \begin{array}{ll} z \in_{\mathit{List}} \mathit{flatten}\ (\mathit{to\text{-}search\text{-}tree}\ \mathit{xs}) & \leftrightarrow \\ z \in_{\mathit{Tree}} \mathit{to\text{-}search\text{-}tree}\ \mathit{xs} & \leftrightarrow \\ z \in_{\mathit{List}} \mathit{xs} & \Box \end{array} \end{array}$$

8 Set Equivalence, Subsets and Subbags

It is easy to tweak the definition of bag equivalence so that we get set equivalence:

$$\begin{array}{l} \_\approx_{set}\_ \;:\; \mathit{List}\ A \to \mathit{List}\ A \to \mathit{Set} \\ xs\ \approx_{set}\ ys\ =\ \forall\ z\ \ \to\ \ z \in xs \Leftrightarrow z \in ys \end{array}$$

This definition states that *xs* and *ys* are set equivalent if, for any value z, z is a member of *xs* iff it is a member of *ys*. We can also define subset and subbag relations:

$$\begin{array}{ll} \_\lesssim_{set}\_ \;:\; \mathit{List}\ A \to \mathit{List}\ A \to \mathit{Set} & \_\lesssim_{bag}\_ \;:\; \mathit{List}\ A \to \mathit{List}\ A \to \mathit{Set} \\ xs\ \lesssim_{set}\ ys\ = & xs\ \lesssim_{bag}\ ys\ = \\ \quad \forall\ z\ \ \to\ \ z \in xs \to z \in ys & \quad \forall\ z\ \ \to\ \ z \in xs \rightarrowtail z \in ys \end{array}$$

Here $A \rightarrowtail B$ stands for the type of injections from A to B: *xs* is a subbag of *ys* if every element occurs at least as often in *ys* as in *xs*.

It is now easy to generalise over the kind of function space used in the four definitions and define $xs \sim[\ k\ ]\ ys$, meaning that *xs* and *ys* are k-related, where k ranges over subset, set, subbag and bag. Using this definition one can prove many preservation properties uniformly for all four relations at once (given suitable combinators, some of which may not be defined uniformly). Here is one example of such a preservation property:

$$\begin{array}{l} \gg\!=\text{-}\mathit{cong}\ :\ (xs\ ys\ :\ \mathit{List}\ A)\ (f\ g\ :\ A \to \mathit{List}\ B) \to \\ \qquad\qquad\quad xs \sim[\ k\ ]\ ys \to (\forall\ x \to f\ x \sim[\ k\ ]\ g\ x) \to xs \gg\!= f \sim[\ k\ ]\ ys \gg\!= g \end{array}$$

Details of these constructions are not provided in the paper due to lack of space. See the accompanying code for more information.

9 Related Work

Morris [15] defines *Any* for arbitrary indexed strictly positive types. The dual of *Any*, *All*, goes back at least to Hermida and Jacobs [8], who define it for polynomial functors. In Hoogendijk and de Moor's treatment of containers [10] membership is a lax natural transformation, and this implies that the following variant of *Any-map* (with $\_\Leftrightarrow\_$ rather than $\_\leftrightarrow\_$) holds: $x \in \mathit{map}\ f\ ys \Leftrightarrow \exists\ \lambda\ y \to x \equiv f\ y\ \times\ y \in ys$.

In a previous paper I used the definitions of bag and set equivalence given above in order to state and formally prove properties of a parser combinator library [6]. That paper did not discuss bijectional reasoning, did not discuss alternative definitions of bag equivalence such as $\_\approx'_{bag}\_$, and did not define bag and set equivalence for arbitrary containers, so the overlap with the present paper is very small. The paper did define something resembling bag and set equivalence for parsers. Given that $x \in p \cdot s$ means that x is one possible result of applying the parser p to the string s we can define the relations as follows:

$p_1 \approx p_2 = \forall\, x\, s \;\rightarrow\; x \in p_1 \cdot s \;\sim\; x \in p_2 \cdot s$. When $\sim$ is $\Leftrightarrow$ we get *language equivalence*, and when it is $\leftrightarrow$ we get the stronger notion of *parser equivalence*, which distinguishes parsers that exhibit differing amounts of ambiguity. Correctness of the *parse* function, which takes a parser and an input string to a list of results, was stated as follows: $x \in p \cdot s \;\leftrightarrow\; x \in \mathit{parse}\ p\ s$. Notice the flexibility provided by the use of bijections: the two sides of the correctness statement refer to different things—an inductive definition of the semantics of parsers to the left, and list membership to the right—and yet they can be usefully related.

Abbott et al. [2] define bags using *quotient containers*. A quotient container is a container $S \rhd P$ plus, for each shape s, a set $G\ s$ of automorphisms on $P\ s$, containing the identity and closed under composition and inverse. Quotient containers are interpreted as ordinary containers, except that the position-to-value functions of type $P\ s \rightarrow A$ (for some A) are quotiented by the equivalence relation that identifies f_1 and f_2 if $f_2 = f_1 \circ g$ for some $g : G\ s$. Abbott et al. define bags by taking the list container $\mathbb{N} \rhd \mathit{Fin}$ and letting $G\ n$ be the symmetric group on $\mathit{Fin}\ n$: $G\ n = \mathit{Fin}\ n \leftrightarrow \mathit{Fin}\ n$. The position-to-value functions of $\mathbb{N} \rhd \mathit{Fin}$ correspond to the *lookup* function, so this definition of bags is very close to what you get if you quotient lists by $\_\approx'_{bag}\_$, the alternative definition of bag equivalence given in Sect. 3. Quotient containers only allow us to identify values which have the same shape, so one could not define bags by starting from the binary tree container defined in Sect. 6 and turning this into a quotient container, at least not in an obvious way.

In the SSReflect [7] library bag equivalence (for finite lists containing elements with decidable equality) is defined as a boolean-valued computable function: the list xs is a permutation of ys if, for every element z of $xs +\!\!+\, ys$, the number of occurrences of z in xs is equal to the number of occurrences in ys.

The Coq [19] standard library contains (at least) two definitions related to bag equivalence. A multiset containing values of type A, where A comes with decidable equality, is defined as a function of type $A \rightarrow \mathbb{N}$, i.e. as a function associating a multiplicity with every element. There is also an inductive definition of bag equivalence which states (more or less) that xs and ys are bag equivalent if xs can be transformed into ys using a finite sequence of transpositions of adjacent elements. It is easy to tweak this definition to get set equivalence, but it does not seem easy to generalise it to arbitrary containers.

Contejean [5] defines bag equivalence for lists inductively by, in effect, enumerating where every element in the left list occurs in the right one. It seems likely that this definition can be adapted to streams, but it is not obvious how to generalise it to branching structures such as binary trees.

In the Boom hierarchy (attributed to Boom by Meertens [14]) the type of bags containing elements of type A is defined as the free commutative monoid on A, i.e. bags are lists where the append operation is taken to be commutative. The type of sets is defined by adding the requirement that the append operation is idempotent. Generalising to types with infinite values seems nontrivial. Hoogendijk [11] and Hoogendijk and Backhouse [12], working with the Boom hierarchy in a relational setting, prove various laws related to bags and sets (as well as lists and binary trees). One result is that the map function preserves bag and set equivalence.

Yorgey [21] points out that one can define the type of bags as a certain (finitary) combinatorial species [13]. A species is an endofunctor in the category of finite sets and bijections; one can see the endofunctor as mapping a set of position labels to a labelled structure. Bags correspond to the species which maps a set A to the singleton set $\{ A \}$, and lifts a bijection $A \leftrightarrow B$ in the obvious way.

10 Conclusions

Through a number of examples, proofs and generalisations I hope to have shown that the definition of bag equivalence presented in this paper is useful. I do not claim that this definition is always preferable to others. For instance, in the absence of proof-irrelevance it seems to be easier to prove that cons is left cancellative using the definition $\_\approx'_{bag}\_$ from Sect. 3 (see the accompanying code). However, $\_\approx_{bag}\_$ and $\_\approx'_{bag}\_$ are equivalent, so in many cases it should be possible to use one definition in one proof and another in another.

As mentioned above I have been careful not to use the K rule when formalising this work. The reason is the ongoing work on homotopy type theory [20], a form of type theory where equality of types is (equivalent to) isomorphism and equality of functions is pointwise equality. With this kind of type theory bag equivalence can be stated as $xs \approx_{bag} ys = (\lambda\ z \rightarrow z \in xs) \equiv (\lambda\ z \rightarrow z \in ys)$, the bijectional reasoning in this paper can be turned into equational reasoning, and preservation lemmas like $+$-*cong* do not need to be proved (because equality is substitutive). However, homotopy type theory is incompatible with the K rule, which implies that all proofs of $A \equiv B$ are equal: the equalities corresponding to the identity function and the not function should be distinct elements of $Bool \equiv Bool$.

Acknowledgements. I would like to thank Simon Thompson and Bengt Nordström for suggesting that my definition of bag and set equivalence could be generalised further, to include other relations; and Fritz Henglein for suggesting that I should use the term "bag equivalence" rather than "bag equality". I would also like to thank Thierry Coquand for suggesting that *Any* could be defined as an iterated sum rather than as an inductive family, and for urging me to make sure that the results are valid also in the absence of the K rule. Finally I would like to thank several anonymous reviewers for useful feedback. In particular, one reviewer suggested that I could use tree sort as an example.

Large parts of this work were done when I was working at the University of Nottingham, with financial support from EPSRC (grant code: EP/E04350X/1). I have also received support from the ERC: "The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 247219."

References

[1] Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theoretical Computer Science 342, 3–27 (2005)

[2] Abbott, M., Altenkirch, T., Ghani, N., McBride, C.: Constructing Polymorphic Programs with Quotient Types. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 2–15. Springer, Heidelberg (2004)

[3] Altenkirch, T., Levy, P., Staton, S.: Higher-Order Containers. In: Ferreira, F., Löwe, B., Mayordomo, E., Mendes Gomes, L. (eds.) CiE 2010. LNCS, vol. 6158, pp. 11–20. Springer, Heidelberg (2010)

[4] Altenkirch, T., Morris, P.: Indexed containers. In: LICS 2009, pp. 277–285 (2009)

[5] Contejean, E.: Modeling Permutations in Coq for Coccinelle. In: Comon-Lundh, H., Kirchner, C., Kirchner, H. (eds.) Rewriting, Computation and Proof. LNCS, vol. 4600, pp. 259–269. Springer, Heidelberg (2007)

[6] Danielsson, N.A.: Total parser combinators. In: ICFP 2010, pp. 285–296 (2010)

[7] Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Tech. Rep. inria-00258384, version 10, INRIA (2011)

[8] Hermida, C., Jacobs, B.: An Algebraic View of Structural Induction. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 412–426. Springer, Heidelberg (1995)

[9] Hofmann, M., Streicher, T.: The groupoid model refutes uniqueness of identity proofs. In: LICS 1994, pp. 208–212 (1994)

[10] Hoogendijk, P., de Moor, O.: Container types categorically. Journal of Functional Programming 10(2), 191–225 (2000)

[11] Hoogendijk, P.F.: (Relational) Programming Laws in the Boom Hierarchy of Types. In: Bird, R.S., Morgan, C.C., Woodcock, J.C.P. (eds.) MPC 1992. LNCS, vol. 669, pp. 163–190. Springer, Heidelberg (1993)

[12] Hoogendijk, P.F., Backhouse, R.C.: Relational programming laws in the tree, list, bag, set hierarchy. Science of Computer Programming 22(1-2), 67–105 (1994)

[13] Joyal, A.: Une théorie combinatoire des séries formelles. Advances in Mathematics 42(1), 1–82 (1981)

[14] Meertens, L.: Algorithmics: Towards programming as a mathematical activity. In: Mathematics and Computer Science, CWI Monographs, vol. 1, pp. 289–334. North-Holland (1986)

[15] Morris, P.W.J.: Constructing Universes for Generic Programming. Ph.D. thesis, The University of Nottingham (2007)

[16] Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology and Göteborg University (2007)

[17] Streicher, T.: Investigations Into Intensional Type Theory. Habilitationsschrift, Ludwig-Maximilians-Universität München (1993)

[18] The Agda Team: The Agda Wiki (2012), http://wiki.portal.chalmers.se/agda/

[19] The Coq Development Team: The Coq Proof Assistant Reference Manual, Version 8.3pl3 (2011)

[20] Voevodsky, V.: Univalent foundations project - a modified version of an NSF grant application (2010) (unpublished)

[21] Yorgey, B.A.: Species and functors and types, oh my! In: Haskell 2010, pp. 147–158 (2010)

Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm

Peter Lammich and Thomas Tuerk

TU München
{peter.lammich,thomas.tuerk}@in.tum.de

Abstract. We provide a framework for program and data refinement in Isabelle/HOL. It is based on a refinement calculus for monadic expressions and provides tools to automate canonical tasks such as verification condition generation. It produces executable programs, from which Isabelle/HOL can generate verified, efficient code in various languages, including Standard ML, Haskell and Scala.

In order to demonstrate the practical applicability of our framework, we present a verified implementation of Hopcroft's algorithm for automata minimisation.

1 Introduction

When verifying algorithms, there is a trade-off between the abstract version of the algorithm, that captures the algorithmic ideas, and the actual implementation, that exploits efficient data structures and other kinds of optimisations. While the abstract version has a clean correctness proof, it is usually inefficient or not executable at all. On the other hand, the correctness proof of the implementation version is usually cluttered with implementation details that obfuscate the main ideas and may even render proofs of more complex algorithms unmanageable. A standard solution to this problem is to *stepwise refine* [3] the abstract algorithm to its implementation, showing that each refinement step preserves correctness. A special case is data refinement [16], which replaces operations on abstract datatypes (e. g. sets) by corresponding operations on concrete datatypes (e. g. red-black trees). In Isabelle/HOL [27], there is some support for data refinement during code generation [14]. However, it is limited to operations that can be fully specified on the abstract level. This limitation is an inherent problem, as the following example illustrates:

Example 1. Consider an operation that selects an arbitrary element from a nonempty set S. Ideally, we would like to write $\varepsilon\ x.\ x \in S$ in the abstract algorithm, where ε is the Hilbert-choice operator. However, this already over-specifies the operation, as one cannot prove that an implementation of this operation actually returns the same element as the choice operator does.

A common solution for this problem is nondeterminism. For the functional language of Isabelle/HOL, this means to use relations instead of functions. However, in order to make refinement of relational programs manageable, some tool support is required. Our first approach in this direction was a formalisation of relational WHILE-loops [20]. It was successfully used to verify a depth first search algorithm [20] and basic tree automata operations [21]. However, it is tailored to algorithms with a single while loop and involves some lengthy boilerplate code.

In contrast, this paper presents a general framework for nondeterministic (relational) programs in Isabelle/HOL. Programs are represented by a monad [35], on which we define a refinement calculus [4,33]. Monads allow a seamless and lightweight integration within the functional logic of Isabelle/HOL. Moreover, we implemented proof tools that automate canonical tasks such as verification condition generation. Thus, the user of our framework can focus on verifying the algorithmic ideas of the program, rather than coping with the details of the underlying refinement calculus. Our framework also integrates nicely with the Isabelle Collection Framework (ICF) [20,19], which makes many verified, efficient data structures readily available. When an algorithm has been refined to its implementation, the code generator of Isabelle/HOL [13,15] produces efficient code in various languages, including Standard ML, Haskell and Scala.

We have conducted several case studies that demonstrate the practical applicability of our framework: The framework itself [22] comes with a collection of simple examples and a userguide that helps one getting started. A more complex development is the formalisation of Dijkstra's shortest path algorithm [28]. In this paper, we present the largest case study so far: As part of the *Computer Aided Verification of Automata*-project[1], we successfully verified Hopcroft's minimisation algorithm [17] for finite automata. The correctness proof of this algorithm is non-trivial even at an abstract level. Moreover, efficient implementations usually use some non-obvious optimisations, which would make a direct correctness proof unmanageable. Thus, it is a good candidate for using stepwise refinement, which allows a clean separation between the algorithmic idea and the optimisations.

Related Work. Data refinement dates back to Hoare [16]. Refinement calculus was first proposed by Back [3] for imperative programs and has been subject to extensive research. Good overviews are [4,11]. There are various mechanised formalisations of refinement calculus for imperative programs (e. g. [23,34,31]). They require quite complicated techniques like window reasoning. Moreover, formalisation of an universal state space, as required for imperative programs, is quite involved in HOL (cf. [32]). As also observed in [9], these complications do not arise in a monadic approach. Schwenke and Mahony [33] combine monads with refinement calculus. However, their focus is different: While we aim at obtaining a simple calculus suited to our problems, they strive for introducing more advanced concepts like angelic nondeterminism. Moreover, they do not treat data refinement. The work closest to ours is the refinement monad used in the seL4 project [9]. While they use a state monad to model kernel operations with side-effects, we focus on refinement of model-checking algorithms, where a

[1] See `http://cava.in.tum.de/`

simpler monad without state is sufficient. In some cases, deterministic specifications and parametrisation may be viable alternatives to relational specifications (cf. [24,14]). However, the relational approach is more general and scalable.

Despite being a widely-used, non-trivial algorithm, we are aware of only one other formalisation of Hopcroft's algorithm, which was performed by Braibant and Pous [8] in Coq. However, they do not verify Hopcroft's algorithm, but a simplified one. The level of abstraction used there is a compromise between having a correctness proof of manageable size and allowing code generation. In addition, there is a formalisation of the Myhill/Nerode theorem in Nuprl by Constable, Jackson, Naumov and Uribe [10]. This formalisation allows the extraction of an inefficient minimisation algorithm.

Contents. The rest of this paper is organised as follows: In Section 2, we describe our refinement framework. In Section 3, a verified, efficient implementation of Hopcroft's algorithm is presented as a case study. Finally, Section 4 provides a conclusion and sketches current and future research.

2 The Refinement Framework

2.1 Basic Concepts

To describe nondeterministic programs, we use a monad over the datatype $(a)M ::= \textbf{res}\ (a\ \mathsf{set}) \mid \textbf{fail}$. Here, a denotes the type of values, and a set denotes sets of values. Intuitively, a result of the form $\textbf{res}\ X$ means that the program nondeterministically returns some value $x \in X$. The result **fail** means that there exists an execution where some assertion failed. On results, we define the ordering $\sqsubseteq$ by $\textbf{res}\ X \sqsubseteq \textbf{res}\ Y$ iff $X \subseteq Y$ and $m \sqsubseteq \textbf{fail}$ for all $m \in (a)M$. If we have $m \sqsubseteq m'$, we say that m *refines* m'. Intuitively, $m \sqsubseteq m'$ means that all values in m are also in m'.

With this ordering, $(a)M$ is a complete lattice with least element $\textbf{res}\ \emptyset$ and greatest element **fail**. The return and bind operations are defined as follows:

$$\textbf{return}\ x := \textbf{res}\ \{x\} \quad \textbf{bind}\ (\textbf{res}\ X)\ f := \bigsqcup(f\ X) \quad \textbf{bind}\ \textbf{fail}\ f := \textbf{fail}.$$

Note that our monad is a combination of the set and exception monad [35]. Intuitively, the **return** x operation returns a single value x, and the **bind** m f operation nondeterministically chooses a value from m and applies f to it. If f fails for some value in m, the operation **bind** m f also fails. We use a **do**-notation similar to Haskell, e. g. we write $\textbf{do}\{x{\leftarrow}M; \textbf{let}\ y = t; f\ x; g\ y\}$ for $\textbf{bind}\ M\ (\lambda x.\ \textbf{let}\ y = t\ \textbf{in}\ \textbf{bind}\ (f\ x)\ (\lambda\_.\ g\ y))$.

A program is a function $f : a \to (r)M$ from argument values to results. Correctness is defined by refinement: for a precondition $\Phi : a \to \mathbb{B}$ and a postcondition $\Psi : a \to r \to \mathbb{B}$, we define $\models \{\Phi\}\ f\ \{\Psi\} := \forall x.\ \Phi\ x \Longrightarrow f\ x \sqsubseteq \textbf{spec}\ (\Psi\ x)$, where we use **spec** synonymously for **res** to emphasise that a set of values is used as specification. Identifying sets with their characteristic predicates, we also use the notation $\textbf{spec}\ x.\ \Phi\ x := \textbf{res}\ \{x \mid x \in \Phi\}$. In this context, **fail** is the result that refines no specification. Dually, $\textbf{res}\ \emptyset$ is the result that refines any specification. We use $\textbf{succeed} := \textbf{res}\ \emptyset$ to emphasise this duality.

2.2 Data Refinement

When deriving a concrete program from an abstract one, we also want to replace abstract data types (e. g. sets) with concrete ones (e. g. hashtables). This process is called *data refinement* [16]. To formalise data refinement, we use an *abstraction relation* $R \subseteq c \times a$ that relates concrete values c with abstract values a. We implicitly assume that R can be written as $R = \{(c, \alpha_R\ c) \mid I_R\ c\}$, where I_R is called *data type invariant* and α_R is called *abstraction function*. This restricts abstraction relations to those relations that map a concrete value to at most one abstract value, which is a natural assumption in our context.

Example 2. Sets of integers can be implemented by lists with distinct elements. The invariant I_R asserts that the list contains no duplicates and the abstraction function α_R maps a list to the set of its values.

This example illustrates two properties of data refinement: First, the invariant is required to sort out invalid concrete values, in our case lists with duplicate elements. Second, an implementation is not required to be complete w. r. t. the abstract data type. In our case, lists can only implement finite sets, while the abstract data type may represent arbitrary sets.

We define a *concretisation function* $\Downarrow R : (a)M \to (c)M$ that maps a result over abstract values to a result over concrete values:

$$\Downarrow R\ (\mathbf{res}\ X') := \mathbf{res}\ \{x \mid I_R\ x \wedge \alpha_R\ x \in X'\} \qquad \Downarrow R\ \mathbf{fail} := \mathbf{fail}$$

Intuitively, $\Downarrow R\, m'$ is the greatest concrete result that corresponds to the abstract result m'. In Example 2, we have $\Downarrow R\ (\mathbf{return}\ \{1,2\}) = \mathbf{res}\ \{[1,2],[2,1]\}$, i. e. the set of all distinct lists representing the set $\{1,2\}$.

We also define an *abstraction function* $\Uparrow R : (c)M \to (a)M$ by

$$\Uparrow R\ (\mathbf{res}\ X) := \begin{cases} \alpha_R\ X & \text{if } \forall x \in X.\ I_R\ x \\ \mathbf{fail} & \text{otherwise} \end{cases} \qquad \Uparrow R\ \mathbf{fail} := \mathbf{fail}$$

We have $\Uparrow R\, m \sqsubseteq m' \Leftrightarrow m \sqsubseteq \Downarrow R\, m'$, i. e. $\Uparrow R$ and $\Downarrow R$ are a *Galois-connection* [25]. Galois-connections are commonly used in data refinement (cf. [26,5]). We exploit their properties for proving the rules for our recursion combinators.

To improve readability, we define the notations

$$m \sqsubseteq_R m' := m \sqsubseteq \Downarrow R\ m' \ \text{ and } \ f \sqsubseteq_{R_a \to R_r} f' := \forall (x, x') \in R_a.\ f\ x \sqsubseteq_{R_r} f'\ x'.$$

Intuitively, $m \sqsubseteq_R m'$ means that the concrete result m refines the abstract result m' w. r. t. the abstraction relation R; $f \sqsubseteq_{R_a \to R_r} f'$ means that the concrete program f refines the abstract program f', where R_a is the abstraction relation for the arguments and R_r the one for the results.

The operators $\sqsubseteq_R$ and $\sqsubseteq_{R_a \to R_r}$ are transitive in the following sense:

$$m \sqsubseteq_{R_1} m' \wedge m' \sqsubseteq_{R_2} m'' \Longrightarrow m \sqsubseteq_{R_1 R_2} m''$$
$$f \sqsubseteq_{R_{a1} \to R_{r1}} f' \wedge f' \sqsubseteq_{R_{a2} \to R_{r2}} f'' \Longrightarrow f \sqsubseteq_{R_{a1} R_{a2} \to R_{r1} R_{r2}} f''$$

where $R_1 R_2 := \{(x, x'') \mid \exists x'.\ (x, x') \in R_1 \wedge (x', x'') \in R_2\}$ is relational composition. This allows stepwise refinement, which is essential for complex programs, such as our case study presented in Section 3.

The following rule captures the typical course of program development with our framework:

$$\begin{aligned} & \models \{\Phi\}\ f_1\ \{\Psi\} \wedge f_n \sqsubseteq_{R_a \to R_r} f_1 \\ \Longrightarrow & \models \{\lambda x.\ I_{R_a}\ x \wedge \Phi\ (\alpha_{R_a}\ x)\}\ f_n\ \{\lambda x\ x'.\ I_{R_r}\ x' \wedge \Psi\ (\alpha_{R_a}\ x)\ (\alpha_{R_r}\ x')\} \end{aligned}$$

First, an initial program f_1 is shown to be correct w. r. t. a specification. Then, it is refined (possibly in many steps) to a program f_n. The conclusion of the rule states that the refined program is correct w. r. t. a specification that results from the abstract specification and the refinement relation: the precondition requires the argument x to satisfy its data type invariant I_{R_a} and the abstraction of the argument $\alpha_{R_a}\ x$ to satisfy the abstract precondition Φ. Then, the postcondition guarantees that the result x' satisfies the data type invariant I_{R_r} and its abstraction $\alpha_{R_r}\ x'$ satisfies the abstract postcondition Ψ.

Ideally, f_1 captures the basic ideas of the algorithm on an abstract level, and f_n uses efficient data structures and is executable. In our context, executable means that the Isabelle code generator [13,15] can generate code for the program. This is the case if the program is deterministic and the expressions used in the program are executable. For technical reasons, we require $f_n : a \to (r)M$ to have the form $\lambda x.$ **return** $(f_{\mathsf{plain}}\ x)$ (for total correct programs) or $\lambda x.\ \mathsf{nres\text{-}of}\ (f_{\mathsf{det}}\ x)$ (for partial correct programs). Here, $f_{\mathsf{plain}} : a \to r$ is a plain function that does not use the result monad at all, and $f_{\mathsf{det}} : a \to (r)M_{\mathsf{det}}$ is a function over a deterministic result monad, which is embedded into the nondeterministic monad via the function $\mathsf{nres\text{-}of} : (a)M_{\mathsf{det}} \to (a)M$. For deterministic programs f_{n-1}, the framework automatically generates f_n and proves that $f_n \sqsubseteq f_{n-1}$ holds.

2.3 Combinators

In this section, we describe the combinators that are used as building blocks for programs. Note that due to the shallow embedding, our framework is extensible, i. e. new combinators can be added without modifying the existing code.

We already covered the monad operations **return** and **bind**, as well as the results **fail** and **succeed**. Moreover, standard constructs provided by Isabelle/HOL can be used (e. g. if, let, case, λ-abstraction).

Assertions. A useful combinator is **assert** $\Phi :=$ **if** Φ **then return** () **else fail**, where () is the element of the unit type. Its intended use is **do**$\{$**assert** $\Phi; m\}$. Assertions are used for refinement in context (cf. [4, Chap. 28]), as illustrated by the following example:

Example 3. Reconsider Example 2, where sets are implemented by distinct lists. This time we select an arbitrary element from a set, i. e. we implement the specification **spec** $x.\ x \in S$ for some nonempty set S. The concrete operation is implemented by the hd-function, which returns the first element of a list. We obviously have $S \neq \emptyset \wedge (l, S) \in R \Longrightarrow \textbf{return}\ \mathit{hd}\ l \sqsubseteq \textbf{spec}\ x.\ x \in S$. However, to apply this rule, we have to know that S is, indeed, nonempty. For this purpose, we use $\textbf{do}\{\textbf{assert}\ S \neq \emptyset; \textbf{spec}\ x.\ x \in S\}$ as the abstract operation. Thus, non-emptiness is shown during the correctness proof of the abstract program, and refinement can be proved under the assumption that S is not empty.

Recursion. In our lattice-theoretic framework, recursion is naturally modelled as a fixed point of a monotonic functional. Functionals built from monad operations are always monotonic and already enjoy some tool support in Isabelle/HOL [18].

Assume that we want to define a function $f : a \rightarrow (r)M$ according to the recursion equation $f\ x = \mathsf{B}\ f\ x$, where $\mathsf{B} : (a \rightarrow (r)M) \rightarrow a \rightarrow (r)M$ is a monotonic functional describing the body of the function. To reason about partial correctness, one naturally chooses the least fixed point of B, i. e. one defines $f := \mathsf{lfp}\ \mathsf{B}$ (cf. [4][2]). To express the monotonicity assumption, we define

$$\textbf{rec}\ \mathsf{B}\ x := \textbf{do}\{\textbf{assert}\ \mathsf{mono}\ \mathsf{B}; \mathsf{lfp}\ \mathsf{B}\ x\}.$$

Intuitively, this definition ignores nonterminating executions. In the extreme case, when there is no terminating execution for argument x at all, we get $\textbf{rec}\ \mathsf{B}\ x = \textbf{succeed}$, which trivially satisfies any specification.

Dually, total correctness is naturally described by the greatest fixed point:

$$\textbf{rec}_{\mathbf{T}}\ \mathsf{B}\ x := \textbf{do}\{\textbf{assert}\ \mathsf{mono}\ \mathsf{B}; \mathsf{gfp}\ \mathsf{B}\ x\}.$$

Intuitively, already a single non-terminating execution from argument x results in $\textbf{rec}_{\mathbf{T}}\ \mathsf{B}\ x = \textbf{fail}$, which trivially does not satisfy any specification.

Note, that the intended correctness property (partial or total) is determined by the combinator (**rec** or $\textbf{rec}_{\mathbf{T}}$) used in the program. It would be more natural to determine this by the specification. The Egli-Milner order [12,30] provides a tool for that. Using it in our framework is left for future work.

From the basic recursion combinators, we derive the combinators **while**, $\textbf{while}_{\mathbf{T}}$ and **foreach**, which model tail-recursive functions and iteration over the elements of a finite set, respectively. These combinators allow for more readable programs and proofs. Moreover, a **foreach**-loop can easily be refined to a fold operation on the data structure implementing the set, which usually results in more efficient programs. The Isabelle Collection Framework [19] provides such fold-functions (called *iterators* there), and our refinement framework can automatically refine **foreach**-loops to iterators.

2.4 Proof Rules

In this section, we describe the proof rules provided by our framework. Given a proof obligation $m \sqsubseteq_R m'$, these rules work by syntactically decomposing m and m', generating new proof obligations for the components of m and m'.

[2] The description there [4, Chap. 20] is based on weakest preconditions. Their lattice is dual to ours, i. e. **fail** (called *abort* there) is the least element and **succeed** (called *magic* there) is the greatest element. Hence, they take the greatest fixed point for partial correctness and the least fixed point for total correctness.

Typically, a refinement step refines **spec**-statements to their implementations and performs data refinements that preserve the structure of the program. Thus, we distinguish two types of refinement: *Specification refinements* have the form $m \sqsubseteq_R \textbf{spec}\ \Phi$, and *pure data refinements* have the form $m \sqsubseteq_R m'$, where the topmost combinator in m and m' is the same. For each combinator, we provide a proof rule for specification refinement and one for pure data refinement.

Note that in practice not all refinements are specification or pure data refinements. Some common cases, like introduction or omission of assertions, are tolerated by our verification condition generator. For more complex structural changes, we provide a method to convert a refinement proof obligation into a boolean formula, based on pointwise reasoning over the underlying sets. This formula is solved using standard Isabelle/HOL tools. In practice, this works well for refinements that do not involve recursion.

For the basic combinators, we use the following rules:

$$(\forall a.\ \Phi\ a \Longrightarrow f\ a \sqsubseteq \textbf{spec}\ (\Psi\ a)) \Longrightarrow\ \models \{\Phi\}\ f\ \{\Psi\} \qquad \text{(fun-sp)}$$
$$\big(\forall a\ a'.\ (a,a') \in R_a \Longrightarrow f\ a \sqsubseteq_{R_r} f'\ a'\big) \Longrightarrow f \sqsubseteq_{R_a \to R_r} f' \qquad \text{(fun-dr)}$$
$$\big(m \sqsubseteq \textbf{spec}\ x.\ \exists\ x'.\ (x,x') \in R \wedge \Phi\ x'\big) \Longrightarrow m \sqsubseteq_R \textbf{spec}\ \Phi \qquad \text{(spec-dr)}$$
$$(\forall x.\ \Phi\ x \Longrightarrow \Psi\ x) \Longrightarrow \textbf{spec}\ \Phi \sqsubseteq \textbf{spec}\ \Psi \qquad \text{(spec-sp)}$$
$$\Phi\ r \Longrightarrow \textbf{return}\ r \sqsubseteq \textbf{spec}\ \Phi \qquad \text{(ret-sp)}$$
$$(r,r') \in R \Longrightarrow \textbf{return}\ r \sqsubseteq_R \textbf{return}\ r' \qquad \text{(ret-dr)}$$
$$m \sqsubseteq \textbf{spec}\ x.\ (f\ x \sqsubseteq \textbf{spec}\ \Phi) \Longrightarrow \textbf{bind}\ m\ f \sqsubseteq \textbf{spec}\ \Phi \qquad \text{(bind-sp)}$$
$$m \sqsubseteq_{R_1} m' \wedge f \sqsubseteq_{R_1 \to R_2} f' \Longrightarrow \textbf{bind}\ m\ f \sqsubseteq_{R_2} \textbf{bind}\ m'\ f' \qquad \text{(bind-dr)}$$

$$\begin{aligned}&\mathsf{mono}\ \mathsf{B} \wedge \Phi\ x_0 \wedge (\forall f\ x.\ \Phi\ x \wedge \models \{\Phi\}\ f\ \{\Psi\} \Longrightarrow \mathsf{B}\ f\ x \sqsubseteq \textbf{spec}\ (\Psi\ x))\\ \Longrightarrow\ &\textbf{rec}\ \mathsf{B}\ x_0 \sqsubseteq \textbf{spec}\ (\Psi\ x_0)\end{aligned} \qquad \text{(rec-sp)}$$

$$\begin{aligned}&\mathsf{mono}\ \mathsf{B} \wedge \Phi\ x_0 \wedge \mathsf{wf}\ V\\ \wedge\ &\big(\forall f\ x.\ \Phi\ x \wedge \models \{\lambda x'.\ \Phi\ x' \wedge (x',x) \in V\}\ f\ \{\Psi\} \Longrightarrow \mathsf{B}\ f\ x \sqsubseteq \textbf{spec}\ (\Psi\ x)\big)\\ \Longrightarrow\ &\textbf{rec}_{\mathbf{T}}\ \mathsf{B}\ x_0 \sqsubseteq \textbf{spec}\ (\Psi\ x_0)\end{aligned} \qquad \text{(rect-sp)}$$

$$\begin{aligned}&\mathsf{mono}\ \mathsf{B} \wedge (x,x') \in R_a \wedge \big(\forall f\ f'.\ f \sqsubseteq_{R_a \to R_r} f' \Longrightarrow \mathsf{B}\ f \sqsubseteq_{R_a \to R_r} \mathsf{B}'\ f'\big)\\ \Longrightarrow\ &\textbf{rec}\ \mathsf{B}\ x \sqsubseteq_{R_r} \textbf{rec}\ \mathsf{B}'\ x' \wedge \textbf{rec}_{\mathbf{T}}\ \mathsf{B}\ x \sqsubseteq_{R_r} \textbf{rec}_{\mathbf{T}}\ \mathsf{B}'\ x'\end{aligned} \qquad \text{(rec(t)-dr)}$$

Note that, due to space constraints, we omitted the rules for some standard constructs (if, let, case). The rules (fun-sp) and (fun-dr) unfold the shortcut notations for programs with arguments, and (spec-dr) pushes the refinement relation inside a **spec**-statement. Thus, the other rules only need to consider goals of the form $m \sqsubseteq \Downarrow R\ m'$ and $m \sqsubseteq \textbf{spec}\ \Phi$. The (spec-sp)-rule converts refinement of specifications to implication. The (ret-sp) and (ret-dr)-rules handle refinement of return statements. The (bind-sp)-rule decomposes the bind-combinator by generating a nested specification. The (bind-dr)-rule introduces a new refinement relation R_1 for the bound result. The (rec-sp)-rule requires to provide an invariant Φ and to prove that the body of the recursive function is correct, under the inductive assumption that recursive calls behave correctly. The (rect-sp)-rule additionally requires a well-founded[3] relation V, such that the parameters of recursive calls decrease according to V. Note, that these rules resemble the rules used for Hoare-Calculus with procedures (cf. [29] for an overview). The intuition of the (rec-dr)-rule is similar: One has to show that the function bodies are in refinement relation, under the assumption that recursive calls are in relation.

Also the rules for while-loops, which are displayed below, resemble their counterparts from Hoare-Calculus:

$$\begin{aligned} & I\ x_0 \wedge (\forall x.\ I\ x \wedge b\ x \Longrightarrow f\ x \sqsubseteq \mathbf{spec}\ I) \wedge (\forall x.\ I\ x \wedge \neg b\ x \Longrightarrow \Phi\ x) \\ \Longrightarrow\ & \mathbf{while}\ b\ f\ x_0 \sqsubseteq \mathbf{spec}\ \Phi \end{aligned} \qquad \text{(while-sp)}$$

$$\begin{aligned} & \mathsf{wf}\ V \wedge I\ x_0 \wedge (\forall x.\ I\ x \wedge b\ x \Longrightarrow f\ x \sqsubseteq \mathbf{spec}\ (\lambda x'.\ I\ x' \wedge (x', x) \in V)) \\ & \wedge (\forall x.\ I\ x \wedge \neg b\ x \Longrightarrow \Phi\ x) \\ \Longrightarrow\ & \mathbf{while_T}\ b\ f\ x_0 \sqsubseteq \mathbf{spec}\ \Phi \end{aligned} \qquad \text{(whilet-sp)}$$

$$\begin{aligned} & (x_0, x_0') \in R \wedge \big(\forall (x, x') \in R.\ b\ x = b'\ x' \wedge \big(b\ x \Longrightarrow f\ x \sqsubseteq_R f'\ x'\big)\big) \\ \Longrightarrow\ & \mathbf{while}\ b\ f\ x_0 \sqsubseteq_R \mathbf{while}\ b'\ f'\ x_0' \wedge \mathbf{while_T}\ b\ f\ x_0 \sqsubseteq_R \mathbf{while_T}\ b'\ f'\ x_0' \end{aligned} \qquad \text{(while(t)-dr)}$$

Verification Condition Generator. The proof rules presented above are engineered such that their iterated application decomposes a refinement proof obligation into new proof obligations that do not contain combinators any more. We implemented a verification condition generator (VCG) that automates this process. Invariants and well-founded relations are specified interactively during the proof. We also provide versions of while-loops that are annotated with their invariant. To also make this invariant available for refinement proofs, the annotated while-loops assert its validity. The rules for the recursion combinators introduce monotonicity proof obligations, which our VCG discharges automatically, exploiting monotonicity of monad expressions (cf. [18]).

When data-refining the bind-combinator (bind-dr), a refinement relation R_1 for the bound result is required. Here, we provide a heuristics that guesses an adequate relation from its type, which works well for most cases. In those cases where it does not work, refinement relations can be specified interactively.

3 Hopcroft's Algorithm

3.1 Motivation

As part of the *Computer Aided Verification of Automata*-Project[4] a library for finite automata is currently developed in Isabelle/HOL [27]. A minimisation algorithm is an important part of such a library. First Brzozowski's minimisation algorithm [36] was implemented, because its verification and implementation is

[3] We define $\mathsf{wf}\ V$ iff there is no infinite sequence $\langle x_i \rangle_{i \in \mathbb{N}}$ with $\forall i.\ (x_{i+1}, x_i) \in V$.

[4] See http://cava.in.tum.de/

straightforward. As the automata library matured, providing Hopcroft's minimisation algorithm [17] was desirable, because it is more efficient.

When we first implemented Hopcroft's algorithm, the refinement framework was not yet available. We therefore used relational WHILE-loops [20]. Unluckily, this technique does not support nondeterministic, nested loops. Therefore, we first verified a very abstract version of Hopcroft's algorithm that does not require an inner loop. In one huge step, this abstract version was then refined to an executable one that uses a deterministic inner loop. Intermediate refinement steps were not possible due to the missing support for nondeterministic, nested loops. A simple refinement step finally led to code generation using the Isabelle Collection Framework [20,19].

Using one large refinement step, resulted in a lengthy, complicated proof. Many ideas were mixed and thereby obfuscated. Another problem was that reusing invariants during refinement was tricky. As a result, some parts of the abstract correctness proof had to be repeated. Worst of all, the implementation of Hopcroft's algorithm used only simple data structures and optimisations. As the refinement proof was already complicated and lengthy, we did not consider it manageable to verify anything more complex. For our examples, the resulting implementation was not significantly faster than the simple, clean implementation of Brzozowski's algorithm. Thus, we did not achieve our goal of providing an efficient minimisation algorithm.

Using the refinement framework solved all these problems. The monadic approach elegantly handles nondeterminism. In particular, nested, nondeterministic loops are supported. Therefore, the huge, cluttered refinement step of the first attempt could be replaced with several small ones. Each of these new refinement steps focuses on only one issue. Hence, they are conceptually much easier and proofs are cleaner and simpler, especially as the refinement framework provides good automation. Constructs like **assert** transport important information between the different layers. This eliminates the need to reprove parts of the invariant. Moreover, these improvements of the refinement infrastructure make it feasible to refine the algorithm further. The generated code behaves well for our examples.

In the following, the new version of the implementation of Hopcroft's algorithm [17] is presented. The presentation focuses on refinement. We assume that the reader is familiar with minimisation of finite automata. An overview of minimisation algorithms can be found in Watson [36].

3.2 Basic Idea

Let $\mathcal{A} = (\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F})$ be a *deterministic finite automaton* (DFA) consisting of a finite set of states $\mathcal{Q}$, an alphabet Σ, a transition function $\delta : \mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$, an initial state q_0 and a set of accepting states $\mathcal{F} \subseteq \mathcal{Q}$. Moreover, let $\mathcal{A}$ contain only reachable states. Hopcroft's algorithm computes the Myhill-Nerode equivalence relation in the form of the partition $\{\{q' \mid q' \text{ is equivalent to } q\} \mid q \in \mathcal{Q}\}$. Given this relation, the minimal automaton can easily be derived by merging equivalent states.

The abstract idea is to first partition states into accepting and non-accepting ones: $\mathsf{part}_{\mathcal{F}} := \{\mathcal{F}, \mathcal{Q} - \mathcal{F}\} - \{\emptyset\}$. The algorithm then repeatedly splits the equivalence classes of the partition and thereby refines the corresponding equivalence relation, until the Myhill-Nerode relation is reached. With the following definitions this leads to the pseudocode shown in Alg. 1. A correctness proof of this abstract algorithm can e. g. be found in [36].

$$\begin{aligned} \mathsf{split}_{\mathcal{A}}(C,\ (a, C_s)) &:= (\ \{q \mid q \in C \wedge \delta(q,a) \in C_s\}, \\ &\qquad\ \{q \mid q \in C \wedge \delta(q,a) \notin C_s\}\ ) \\ \mathsf{splittable}_{\mathcal{A}}(C,\ (a, C_s)) &:= \ \text{let } (C_t, C_f) = \mathsf{split}_{\mathcal{A}}(C, (a, C_s)) \text{ in } C_t \neq \emptyset \wedge C_f \neq \emptyset \end{aligned}$$

```
initialise $\mathcal{P}$ with $\mathsf{part}_{\mathcal{F}}$;
**while** *there are* $C \in \mathcal{P}$ *and* $(a, C_s) \in \Sigma \times \mathcal{P}$ *with* $\mathsf{splittable}_{\mathcal{A}}(C, (a, C_s))$ **do**
    choose such $C$ and $(a, C_s)$;
    update $\mathcal{P}$ by removing $C$ and adding the two results of $\mathsf{split}_{\mathcal{A}}(C, (a, C_s))$;
**end**
return $\mathcal{P}$;
```

Algorithm 1. Pseudocode for the Basic Idea of Hopcroft's Algorithm

3.3 Abstract Algorithm

The tricky part of Alg. 1 is finding the right *splitter* (a, C_s). Hopcroft's algorithm maintains an explicit set L of all splitters that still need to be tried. Two observations are used to keep L small: once a splitter has been tried, it does not need to be retried later. Moreover, given C, C_t, C_f, a, C_s with $\mathsf{split}_{\mathcal{A}}(C,\ (a, C_s)) = (C_t, C_f)$, it is sufficient to consider only two of the three splitters (a, C), (a, C_t), (a, C_f). With some boilerplate to discharge degenerated corner-cases, these ideas lead to Alg. 2.

```
**Hopcroft_step_abstract**$(\mathcal{A}, a, C_s, \mathcal{P}, L)$ =
  **spec** $(\mathcal{P}', L')$. $\mathcal{P}' = \mathsf{Split}_{\mathcal{A}}(\mathcal{P}, (a, C_s)) \ \wedge\ \mathsf{splitter\_P}_{\mathcal{A}}(\mathcal{P}, (a, C_s), L, L')$;

**Hopcroft_abstract**$(\mathcal{A})$ =
  **if** $(\mathcal{Q} = \emptyset)$ **then return** $\emptyset$ **else**
  **if** $(\mathcal{F} = \emptyset)$ **then return** $\{\mathcal{Q}\}$ **else**
  $\mathbf{while}_{\mathbf{T}}^{\mathsf{Hopcroft\_abstract\_invar}}$$(\lambda(\mathcal{P}, L).\ L \neq \emptyset)$ $(\lambda(\mathcal{P}, L).$ **do** {
    $(a, C_s) \leftarrow$ **spec** $x.\ x \in L$;
    $(\mathcal{P}', L') \leftarrow \mathsf{Hopcroft\_step\_abstract}(\mathcal{A}, a, C_s, \mathcal{P}, L)$;
    **return** $(\mathcal{P}', L')$;
  }) $(\mathsf{part}_{\mathcal{F}}, \{(a, \mathcal{F}) \mid a \in \Sigma\})$
```

Algorithm 2. Abstract Hopcroft

Notice, that Alg. 2 is written in the syntax of the refinement framework. Therefore, the formal definition in Isabelle/HOL looks very similar to the one presented here. Due to space limitations, some details are omitted here, though. In particular, the loop invariant Hopcroft_abstract_invar, as well as the formal definitions of Split and splitter_P are omitted[5]. Informally, the function Split splits all equivalence classes of a partition. splitter_P is an abstract specification of the desired properties of the new splitter set L'. It states that every element (a, C) of L' is either a member of L or that C was added to $\mathcal{P}'$ in this step. All splitters (a, C) such that $C \neq C_s$ and C has not been split, remain in the splitter set. For all classes C that are split into C_t and C_f and for all $a \in \Sigma$ at least one of the splitters (a, C_t) or (a, C_f) needs to be added to L'. If (a, C) is in L, both need to be added.

3.4 Set Implementation

In a refinement step, Split and splitter_P are implemented using a loop (see Alg. 3). First a subset $\mathcal{P}'$ of $\mathcal{P}$ is chosen, such that all classes that need to be split are in $\mathcal{P}'$. Then the **foreach** loop processes each class $C \in \mathcal{P}'$ and updates L and $\mathcal{P}$. Hopcroft_step_set is a correct refinement of Hopcroft_step_abstract (see Alg. 2) provided the invariant of Alg. 2 holds. The refinement framework allows us to use this invariant without reproving it.

Alg. 3 is a high-level representation of Hopcroft's algorithm similar to the ones that can be found in literature (e. g. [36]). By fixing a set representation it would be easily possible to generate code. From now on, refinement focuses on performance improvements.

```
Hopcroft_step_set($\mathcal{A}, a, C_s, \mathcal{P}, L$) =
  $\mathcal{P}' \leftarrow$ spec $\mathcal{P}'$. $\mathcal{P}' \subseteq \mathcal{P} \wedge (\forall C \in \mathcal{P}.\ \mathsf{splittable}_{\mathcal{A}}(C, (a, C_s)) \Rightarrow C \in \mathcal{P}')$;
  $(\mathcal{P}', L') \leftarrow$ foreach$^{\mathsf{Hopcroft\_set\_invar}}$ $\mathcal{P}'$ $(\lambda C\ (\mathcal{P}', L')$. do {
    let $(C_t, C_f) = \mathsf{split}_{\mathcal{A}}(C, (a, C_s))$;
    if $(C_t = \emptyset \vee C_f = \emptyset)$ then return $(\mathcal{P}', L')$ else do {
      $(C_1, C_2) \leftarrow$ spec $x$. $x \in \{(C_f, C_t), (C_t, C_f)\}$;
      let $\mathcal{P}' = (\mathcal{P}' - \{C\}) \cup \{C_1, C_2\}$;
      let $L' = (L' - \{(a, C) \mid a \in \Sigma\}) \cup \{(a, C_1) \mid a \in \Sigma\} \cup \{(a, C_2) \mid (a, C) \in L'\}$;
      return $(\mathcal{P}', L')$;
    }
  }) $(\mathcal{P}, L - \{(a, C_s)\})$;
  return $(\mathcal{P}', L')$;
```

Algorithm 3. Hopcroft Step

3.5 Precomputing the Predecessors

The loop of Alg. 3 computes split$(C, (a, C_s))$ for many different classes C, but a fixed splitter (a, C_s). As an optimisation, the set *pre* $:= \{q \mid \delta(q, a) \in C_s\}$ is precomputed. It is used to compute split. Moreover, the choice of C_1 and C_2 is fixed using the cardinality of C_f and C_t. Provided only finite classes are used, the

[5] They are available at http://cava.in.tum.de as part of the Isabelle/HOL sources.

resulting algorithm is a refinement of Alg. 3. Again, the refinement framework allows us to use the invariant of Alg. 2 to derive this finiteness condition.

3.6 Representation of Partitions

Representing partitions as sets of sets leads to inefficient code. Therefore, the following data refinement changes this representation to one based on finite maps. *im* maps an integer index to a class (represented as a set). *sm* maps a state to the index of the class it belongs to. An integer n is used to keep track of the number of classes. The following abstraction function and invariant are used for the resulting triples (im, sm, n):

$$\begin{aligned}&\mathsf{partition\_map\_invar}(im, sm, n) := \\ &\quad \mathsf{dom}(im) = \{i \mid 0 \le i < n\} \ \wedge\ (\forall 0 \le i < n.\ im(i) \neq \emptyset) \wedge \\ &\quad (\forall q.\ sm(q) = i \Leftrightarrow (0 \le i < n \wedge q \in im(i)))\end{aligned}$$

$$\mathsf{partition\_map\_}\alpha(im, sm, n) := \{im(i) \mid 0 \le i < n\}$$

Using this representation leads to Alg. 4. When splitting a class C, the old index is updated to point to $C_{\max}$. As a result, the update of L becomes much simpler, as the replacement of splitters (a, C) with $(a, C_{\max})$ now happens implicitly. In contrast to previous refinement steps, the data-refinement also requires a straightforward refinement of the outer loop. Due to space limitations, this refinement is not shown here.

```
$\mathsf{Hopcroft\_step\_map}(\mathcal{A}, a, i_s, (im, sm, n), L) =$
  let $pre = \{q \mid \delta(q,a) \in im(i_s)\}$;
  $\mathcal{I} \leftarrow$ spec $\mathcal{I}$. $\mathcal{I} \subseteq \{sm(q) \mid q \in pre\} \wedge$
                 $(\forall q \in pre.\ \mathsf{splittable}_{\mathcal{A}}(im(sm(q)),\ (a, im(i_s))) \Rightarrow sm(q) \in \mathcal{I})$;
  $((im', sm', n'), L') \leftarrow$ foreach$^{\mathsf{Hopcroft\_map\_invar}}$ $\mathcal{I}$ $(\lambda i\ ((im', sm', n'), L')$. do {
    let $(C_t, C_f) = (\{q \mid q \in im(i) \wedge q \in pre\}, \{q \mid q \in im(i) \wedge q \notin pre\})$;
    if $(C_f = \emptyset)$ then return $((im', sm', n'), L')$ else do {
      let $(C_{\min}, C_{\max}) =$ if $(|C_f| < |C_t|)$ then $(C_f, C_t)$ else $(C_t, C_f)$;
      let $(im', sm', n') = (\ im'(i \mapsto C_{\max}, n \mapsto C_{\min}),$
                             $\lambda q.\ if\ q \in C_{\min}\ then\ n\ else\ sm'(q),\ n' + 1)$;
      let $L' = \{(a, n) \mid a \in \Sigma\} \ \cup\ L'$;
      return $((im', sm', n'), L')$;
    }
  }) $((im, sm, n), L - \{(a, i_s)\})$;
  return $((im', sm', n'), L')$;
```

Algorithm 4. Hopcroft Map Representation

3.7 Representation of Classes

Alg. 4 is already executable efficiently. However, refining the representation of classes leads to further performance improvements. Following the implementation of Hopcroft's algorithm by Baclet and Pagetti [6], we use a bijective, finite map pm from $\mathcal{Q}$ to $\{i \mid 0 \leq i < |\mathcal{Q}|\}$ and its inverse pim. By carefully updating pm and pim during the run of the algorithm, it can be ensured that all classes that are needed are of the form $\{pim(i) \mid l \leq i \leq u\}$. Therefore, classes can be represented by a pair of indices l, u. Due to space limitations, details are omitted here[5].

3.8 Code Generation

Thanks to the good integration of our refinement framework with the Isabelle Collection Framework (ICF) [19], it is straightforward to implement the sets and finite maps used in Hopcroft's algorithm with executable datastructures provided by the ICF.

For the implementation we fix states to be natural numbers. This allows us to implement finite maps like *pm*, *pim*, *im* or *sm* by arrays. Other finite maps as well as most sets are implemented by red-black-trees.

For computing the set of predecessors *pre*, it is vital to be able to efficiently iterate over the union of many sets. This is achieved by using sorted lists of distinct elements for computing *pre*. In order to efficiently look up the set of predecessors for a single state and label, a datastructure using arrays and sorted lists is generated in a preprocessing step.

The set of splitters is implemented by an unsorted list of distinct elements. Lists can easily be used as a stack and then provide the operations of inserting a new element and removing an element very efficiently. More importantly though, the choice of the splitter influences the runtime considerably. Experiments by Baclet and Pagetti [6] suggest that implementing L as a stack is a good choice.

Once the datastructures have been instantiated, Isabelle is able to generate code in several programming languages including SML, OCaml, Haskell and Scala.

3.9 Experimental Results

To test our implementation, we benchmarked it against existing implementations of minimisation algorithms. We compare our implementation with a highly optimised implementation of Hopcroft's algorithm by Baclet and Pagetti [6] in OCaml. Moreover, we compare it with an implementation of Blum's algorithm [7] that is part of an automaton library by Leiß[6] in Standard ML. It is unfortunate for the comparison that Leiß implements Blum's algorithm instead of Hopcroft's. However, we believe that a comparison is still interesting, because Blum's algorithm has the same asymptotic complexity. As it was written for teaching, Leiß' library is using algorithms and code that are comparably easy to understand. Keeping this in mind, a bug[7] that was discovered after 10 years in the minimisation algorithm is a strong argument for verified implementations.

For a fair comparison, we generated code in OCaml and PolyML for our implementation and benchmarked it against these two implementations. It is hard

[6] `http://www.cis.uni-muenchen.de/~leiss`

[7] See changelog in `fm.sml`.

to decide though, what exactly to measure. Most critically, Leiß' implementation as well as ours compute the minimal automaton, while the code of Baclet and Pagetti stops after computing the Myhill-Nerode equivalence relation. This is significant, because with our red-black-tree based automata implementation, between 35 and 65 % of the total runtime is spent with constructing the minimal automaton. Implementations based on arrays would be considerably faster for this operation.

Another problem are integers. Baclet and Pagetti use the default integer type of OCaml, which is – depending on the architecture – either 31 or 63 bit wide. In contrast, Leiß' and we use arbitrary size integers. The OCaml implementation of arbitrary size integers is slow. Replacing them leads to a speedup of our OCaml code of a factor of about 2.5, but results in unsound code for huge automata.

Because of these issues, we decided to provide measurements for a version that generates the minimal automaton and uses arbitrary size integers as well as measurements for a version that stops after computing the Myhill-Nerode equivalence relation and uses the default integers of OCaml. A random generator for DFAs [2], which is available as part of the FAdo [1] library, was used to generate sets of benchmark automata. Fig. 1 shows how long each implementation needs to minimise all the automata in these benchmark sets. The numbers in parentheses denote the runtime when stopping after generating the Myhill-Nerode equivalence relation.

No. DFAs	*No. states*	*No. labels*	*Baclet/Pagetti OCaml*	*Lammich/Tuerk OCaml*		*Lammich/Tuerk PolyML*		*Leiß PolyML*
10000	50	2	0.17 s	6.59 s	(1.62 s)	1.88 s	(1.02 s)	5.38 s
10000	50	5	0.27 s	12.62 s	(3.34 s)	3.51 s	(1.83 s)	19.34 s
10000	100	2	0.31 s	14.31 s	(3.30 s)	3.97 s	(1.89 s)	16.41 s
10000	100	5	0.51 s	26.13 s	(6.79 s)	7.56 s	(3.78 s)	63.21 s
10000	250	2	0.69 s	41.02 s	(11.12 s)	11.09 s	(5.17 s)	83.62 s
1000	1000	2	0.51 s	18.61 s	(4.92 s)	5.37 s	(2.36 s)	134.21 s
1000	2500	2	1.44 s	51.35 s	(13.52 s)	17.88 s	(6.89 s)	905.82 s

Fig. 1. Experimental Results (measured on an Intel Core I7 2720QM)

The implementation of Leiß behaves worst, especially on larger automata. This might be partly due to the different algorithm. The implementation by Baclet and Pagetti clearly performs best. It outperforms our implementation roughly by one order of magnitude.

4 Conclusion

We have presented a framework for stepwise refinement of monadic programs in Isabelle/HOL. The stepwise refinement approach leads to a clean separation between the abstract model of an algorithm, which has a nice correctness proof, and the optimisations that eventually yield an efficient implementation. Our framework provides theorems and tools that simplify refinement steps and thus allow the user to focus on the ideas of the algorithm. We have demonstrated the usefulness of our framework by various examples. The most complex ones, verified implementations of Dijkstra's algorithm [28] and Hopcroft's algorithm, would not have been manageable without this framework.

Current and Future Research. A topic of current research is to add even more automation to our framework. For example, we have implemented a prototype tool that automatically refines abstract data types to efficient implementations taken from the Isabelle Collection Framework [19]. The translation is controlled by a map between abstract and concrete types, and optional annotations to resolve ambiguities. The tool works well for examples of medium complexity, like our formalisation of Dijkstra's algorithm [28], but still requires some polishing.

Another interesting topic is to unify our approach with the one used in the seL4 project [9]. The goal is a general framework that is applicable to a wide range of algorithms, including model-checking algorithms and kernel-functions.

Acknowledgements. We thank Manuel Baclet and Thomas Braibant for interesting discussions and their implementations of Hopcroft's algorithm. We also thank Petra Malik for her help with formalising finite automata.

References

1. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: Tools for Automata Manipulation and Visualization. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 65–74. Springer, Heidelberg (2009)
2. Almeida, M., Moreira, N., Reis, R.: Enumeration and generation with a string automata representation. Theor. Comput. Sci. 387, 93–102 (2007)
3. Back, R.J.: On the correctness of refinement steps in program development. PhD thesis, Department of Computer Science, University of Helsinki (1978)
4. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
5. Back, R.J., von Wright, J.: Encoding, decoding and data refinement. Formal Aspects of Computing 12, 313–349 (2000)
6. Baclet, M., Pagetti, C.: Around Hopcroft's Algorithm. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 114–125. Springer, Heidelberg (2006)
7. Blum, N.: An O(n log n) implementation of the standard method for minimizing n-state finite automata. Information Processing Letters 6(2), 65–69 (1996)
8. Braibant, T., Pous, D.: A tactic for deciding kleene algebras. In: First COQ Workshop (2009)
9. 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)
10. Constable, R.L., Jackson, P.B., Naumov, P., Uribe, J.: Formalizing automata theory i: Finite automata (1997)
11. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press (1998)
12. Egli, H.: A mathematical model for nondeterministic computations. Technical report, ETH Zürich (1975)

13. Haftmann, F.: Code Generation from Specifications in Higher Order Logic. PhD thesis, Technische Universität München (2009)
14. Haftmann, F.: Data refinement (raffinement) in Isabelle/HOL (2010), `https://isabelle.in.tum.de/community/`
15. 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)
16. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1, 271–281 (1972), doi:10.1007/BF00289507
17. Hopcroft, J.E.: An $n \log n$ algorithm for minimizing the states in a finite automaton. In: Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
18. Krauss, A.: Recursive definitions of monadic functions. In: Proc. of PAR, pp. 1–13 (2010)
19. 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)
20. Lammich, P.: Collections framework. In: The Archive of Formal Proofs (2009), `http://afp.sf.net/entries/collections.shtml`, Formal proof development
21. Lammich, P.: Tree automata. In: The Archive of Formal Proofs (2009), `http://afp.sf.net/entries/Tree-Automata.shtml`, Formal proof development
22. Lammich, P.: Refinement for monadic programs. In: The Archive of Formal Proofs (2012), `http://afp.sf.net/entries/DiskPaxos.shtml`, Formal Proof Development
23. Langbacka, T., Ruksenas, R., von Wright, J.: Tkwinhol: A Tool for Doing Window Inference in Hol. In: Schubert, E.T., Alves-Foss, J., Windley, P. (eds.) HUG 1995. LNCS, vol. 971, pp. 245–260. Springer, Heidelberg (1995)
24. 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)
25. Melton, A., Schmidt, D., Strecker, G.: Galois Connections and Computer Science Applications. In: Poigné, A., Pitt, D.H., Rydeheard, D.E., Abramsky, S. (eds.) Category Theory and Computer Programming. LNCS, vol. 240, pp. 299–312. Springer, Heidelberg (1986)
26. Müller-Olm, M.: Modular Compiler Verification — A Refinement-Algebraic Approach Advocating Stepwise Abstraction. LNCS, vol. 1283. Springer, Heidelberg (1997)
27. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
28. Nordhoff, B., Lammich, P.: Formalization of Dijkstra's algorithm (2012), Formal Proof Development
29. Olderog, E.R.: Hoare's Logic for Programs with Procedures What has been Achieved? In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 383–395. Springer, Heidelberg (1984)
30. Plotkin, G.D.: A powerdomain construction. SIAM J. Comput. 5, 452–487 (1976)
31. Preoteasa, V.: Program Variables — The Core of Mechanical Reasoning about Imperative Programs. PhD thesis, Turku Centre for Computer Science (2006)
32. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)

33. Schwenke, M., Mahony, B.: The essence of expression refinement. In: Proc. of International Refinement Workshop and Formal Methods, pp. 324–333 (1998)
34. Staples, M.: A Mechanised Theory of Refinement. PhD thesis, 2nd edn. University of Cambridge (1999)
35. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science, pp. 61–78 (1992)
36. Watson, B.W.: A taxonomy of finite automata minimization algorithms. Comp. Sci. Note 93/44, Eindhoven University of Technology, The Netherlands (1993)

Synthesis of Distributed Mobile Programs Using Monadic Types in Coq*

Marino Miculan and Marco Paviotti

Dept. of Mathematics and Computer Science, University of Udine, Italy
marino.miculan@uniud.it, marco.paviotti@gmail.com

Abstract. We present a methodology for the automatic synthesis of certified, distributed, mobile programs with side effects in Erlang, using the Coq proof assistant.

First, we define monadic types in the Calculus of Inductive Constructions, using a lax monad covering the distributed computational aspects. These types can be used for the specifications of programs in Coq. From the (constructive) proofs of these specifications we can extract Haskell code, which is decorated with symbols representing distributed nodes and specific operations for distributed computations. These syntactic annotations are exploited by a *back-end* compiler to produce actual mobile code for a suitable runtime environment (Erlang, in our case).

Then, we introduce an object type theory for distributed computations, which can be used as a front-end programming language. These types and terms are translate to CIC extended with monadic types; this allows us to prove the soundess of the object type theory, and to obtain an implementation of the language via Coq's extraction features.

This methodology can be ported to other computational aspects, by suitably adapting the monadic type theory and the back-end compiler.

1 Introduction

One of the most interesting features of type-theory based proof assistants, like Coq, is the possibility of extracting programs from proofs [9,6]. The main motivation for this extraction mechanism is to produce certified programs: each property proved in Coq (e.g., a precise specification between input and output of programs) will still be valid after extraction. Since the extraction mechanism relies on the well-known Curry-Howard "proofs-as-programs", "propositions-as-types" isomorphism, the extracted programs are naturally expressed in functional languages such as OCaml, Haskell or Scheme.

However, there are many other computational aspects that a programmer has to deal with, and whose programs are quite difficult to reason on; e.g., distributed concurrent programs, web services, mobile code, etc. These scenarios would greatly benefit from mechanisms for extraction and automatic synthesis of certified programs. Unfortunately, the programming languages implementing

* Work supported by MIUR PRIN project 20088HXMYN, "SisteR".

these aspects usually do not feature a rich type theory supporting a Curry-Howard isomorphism. Even if such a theory were available, implementing a specific proof-assistant with its own extraction facilities would be a daunting task.

In this paper, we propose a methodology for circumventing this problem using the extraction mechanisms of existing proof assistants, namely Coq. Basically, the idea is that the proof assistant's type theory (e.g., the Calculus of Inductive Constructions) can be extended with a suitable computational monad `IO : Set -> Set`, covering the specific non-functional computational aspect, similarly to what is done in pure functional languages (e.g., Haskell). These monadic types can be used in the statements of `Propositions` to specify the types of (non-functional) programs, like e.g., `f : A -> IO B`. These propositions can be proved constructively as usual, possibly using the specific properties of the monad operators. From these proofs we can extract functional programs (e.g., in Haskell) by taking advantage of the standard Coq extraction facility. In general these programs cannot be immediately executed because the functional language may not support the specific computational feature we are dealing with. Nevertheless, we can cover this gap by implementing a suitable "post-extraction" translation from Haskell to a suitable target language, with the required features. In fact, the non-functional features in the extracted Haskell programs are represented by the constructors of the computational monad, and these informations can be easily exploited by the post-extraction translation to generate the target program. Thus, Haskell can be seen as an intermediate language, and the post-extraction translation is technically a *back-end compiler*.

Overall, this methodology can be summarized in the following steps:

1. Define a `IO` monad over `Set` in Coq, with the required constructors covering the intended computational aspects.
2. Implement the back-end compiler from Haskell to the target language. Basically this means to define how each constructor of the `IO` monad is actually implemented in the target language. These implementations have to respect the assumptions/the reference implementations.
3. State and constructively prove propositions over types of the form `IO A`. A typical format is the following: `forall x:A, {y:(IO B) | P(x,y)}`. These proofs can use the properties of the monad operators.
4. Extract the Haskell function from the proof, and use the back-end compiler to translate this code into the target language.
5. Execute in the target environment the program obtained in this way.

This approach is quite general and powerful, as it can be effectively used for programming in Coq in different notions of computational. However, it requires the user to have a non trivial knowledge of the Coq proof system. In fact, a programmer may prefer to use a high-level language, possibly specialized *ad hoc* for some particular aspects. Being more specialized, these *front-end* languages are not as expressive as the Calculus of Inductive Constructions, but simpler to use. Still we can consider to translate these languages into Coq, by means of a *front-end compiler*: each syntactic type of the object language is interpreted as a `Set` (possibly using the monad), and terms of the object syntax denote CIC terms between these

`Sets`. Thus, CIC extended with the computational monad is used as a framework for the *semantic* interpretation of a given object type theory. This allows to take advantage of Coq type checking for checking object level terms; moreover, we can obtain readily and implementation of the front-end language by means of the Coq extraction mechanisms and the backend compiler.

Hence, we can extend the above methodology with the following steps:

6. Define a front-end language, i.e., an *object* theory of types and terms. Non-functional computational aspects should be covered by a lax modality.
7. Formalize in Coq the object type theory, the terms, and the typing judgment, via a deep encoding (i.e., as `Inductive Sets`).
8. Define a translation of the object types to Coq `Sets`. In particular, object types with lax modality are interpreted to sets of the form `IO A`.
9. Prove that the type theory is consistent w.r.t. the semantics, giving a translation of well-typed object terms to functions among the corresponding `Sets`.

At this point, we can specify a program using the object type theory; translate this specification into a Coq `Proposition`; prove it as usual; extract the Haskell program, and compile it into the target language. Or we can write a program directly in the object term language, translate it to a Coq proof via the soundess result, extract the Haskell program and compile it to the target language.

In the rest of this paper we will apply this methodology for the synthesis of distributed, mobile programs in Erlang [1,11]. This is particularly relevant since Erlang is an *untyped* language, with little or no support for avoiding many programming errors. On the other hand Erlang offers excellent support for distributed, mobile computations.

First, in Section 2 we introduce the "semantic" monad `IO` in Coq, with corresponding constructors covering the intended computational aspects (namely, code mobility, side-effects and errors). In Section 3 we describe the back-end compiler from Haskell to Erlang, together with a complete working (although quite simple) example. Then, in Section 4 we will define λ_{XD}, a type theory for distributed computations with effects similar to Licata and Harper's HL5 [7], together with its formalization in Coq (using both weak HOAS and de Bruijn indexes at once). We show how these syntactic datatypes and terms are translated into Coq `Sets` (using the `IO` monad) and functions, thus proving the soundness of the object type theory. As a not-so-trivial example, in Section 5 we give a complete specification (in λ_{XD}) and extraction of remote read/write operations. Concluding remarks and directions for further work are in Section 6.

The Coq and Haskell code of the whole development, with examples, is available at `http://sole.dimi.uniud.it/~marino.miculan/LambdaXD/` .

2 Monadic Types for Distributed Computations in Coq

In order to model distributed computation with effects we need to define what a store is, and take into account that we are in a distributed scenario. To this end we define a monad (actually, a world-indexed family of monads) covering all non-functional computational aspects: distributed dependency of resources and memory operations and side effects, possibly with failure.

The type `Store` models distributed stores. Among several possibilites, the simplest is to keep a *global store* which contains all the memory locations of every host. Let us assume to have a type `world` of "worlds", i.e., host identifiers, and for simplicity that locations and values can be only natural numbers. Then, a store is a list of triples "(world, location, value)":

```
Inductive Ref (w : world): Set :=  Loc : nat -> Ref w.
Inductive Store : Set :=
 Empty : Store | SCons : (world * (nat * nat)) -> Store -> Store.
```

We can now define the family of monads: given a world w, for each type A the type `IO w A` is the type of computations returning values in A on world w:

```
Definition Result (w: world) (A: Set): Set := A * Store.
Definition IO (w : world) (A : Set) : Set :=
   Store -> option (Result w A).
```

where `option` is the counterpart of Haskell's `Maybe`. Thus a computation yields either an error or a value of type `Result w A` which contains a value of type A, localized at w, and the new store. Thus, a computation of type `(IO w A)` carries the stores of all worlds, not only w. This is needed for allowing a computation at w to execute computations on other worlds, possibly modifying their stores. As a consequence, a term of type $(\mathsf{Ref} \to \bigcirc A)$ `<w>` actually is a function of type `(Ref w) -> (IO w` A`<w>)`: the argument must be a reference local to world w.

Now we have to define the constructors for the monadic types. The first two constructors are those of any monad, namely "return" and "bind". `IOret` embeds a value as a computation, i.e., a function from states to results; `IObind` "concatenates" the effects.

```
Definition IOret (w : world) (A: Set) (x : A): IO w A :=
  fun (s: Store) => Some (pair x s).
Definition IObind (w:world)(A B:Type)(a:IO w A)(f:A -> IO w B):IO w B :=
  fun s : Store =>
    match (a s) with Some (pair a' s') => (f a') s' | None => None  end.
```

Then, the state is manipulated by the usual "lookup" and "update":

```
Definition IOlookup (w:world)(A:Set):Ref w -> (nat -> IO w A) -> IO w A:=
  fun addr f s =>
    match (do_seek_ref s addr) with Some result => f result s
                                  | None => None   end.
Definition IOupdate (w: world) (A: Set):
Ref w -> nat -> IO w A -> IO w A :=
  fun addr v m s =>  match m s with
                     Some (result, result_state) =>
                        let r := (do_update w result_state addr v) in
                          match r with None => None
                                 |  Some s' => Some (pair result s')
                          end
                    | None => None
    end.
```

where `do_seek_ref` and `do_update` are suitable auxiliary functions. Basically, `(IOlookup w A n f)` first finds the value `v` associated to location n on world w, then executes the computation `f v`; similarly, `(IOupdate w A n v M)` executes M in the state obtaned by updating the location n with the value v. (For sake of simplicity, all references are given a different location number, even if they reside on different worlds.)

New locations are generated by means of `IOnew`, which adds a new local reference at the top of the state, before executing the continuation:

```
Definition IOnew (w:world)(A:Set): nat -> (Ref w -> IO w A) -> IO w A :=
 fun (x : nat) (f: Ref w -> IO w A) (s : State) =>
   let location := (Loc w (IOfree_index w s)) in
     let new_state := (SCons (pair (pair w (IOfree_index w s)) x) s) in
       f location new_state.
```

For modeling distributed computations, we add a function allowing for remote executions and retrieving the value as a local result.

```
Definition IOget (w remote: world) (A: Type) : (IO remote A) -> IO w A :=
  fun (a : IO remote A) (s : Store) => (a s)
```

Given a state `s`, `IOget` function executes the remote computation by giving it the state and returning the same object but in a different world. This could look strange, as the result is untouched—but the type is different, since it becomes `IO w A` (automatically inferred by Coq typing system). Notice that we can pass the remote computation the state of a local one, because each `s: Store` contains the locations and corresponding values for every world.

Remark 1. We have given an explicit definition of the data structures and operations needed for the `IO` monad. A more abstract approach would be to define the operations of the monads as atomic constructors of an abstract data type, together with suitable equational laws defining their behaviour:

```
Record Monad := mkMonad {
  IO : world -> Type -> Type;
  IOreturn : forall w A, A -> (IO w A);
  IObind   : forall w A B, (IO w B) -> (B -> (IO w A)) -> (IO w A);
  IOlookup : forall w A, Ref w -> (nat -> IO w A) -> (IO w A);
  IOupdate : forall w A, Ref w -> nat -> (IO w A) -> (IO w A);
  IO_H1 : forall w A a f, (IObind w A A (IOreturn w A a) f) = (f a);
  ...
}.
```

This approach would be cleaner, since it abstracts from the particular implementation. However, in order to be able to prove all properties we may be interested in, we need to know an equational theory (more precisely, a (Σ, E) algebraic specification) complete with respect to the intended computational aspects. This is a not trivial requirement. Power and Plotkin have provided in [10] a complete specification for the state monad, but we do not know of a similar complete axiomatizations for distributed computations, nor if and how this would “merge” with the former. Hence, we preferred to give an explicit definition for each operation: on one hand, the required properties can be proved instead to be assumed, and on the other these definitions have to be intended as "reference implementations" of the computational aspects.

3 Extraction of Distributed Programs

The monadic types described above, can be used for specifying programs for mobile distributed computation, as in the following example:

Example 1 (Remote procedure call). We prove a Lemma stating that a procedure on a world `server` can be seen as a procedure local to a world `client`:

```
Lemma rpc : forall client server,
            (nat -> (IO server bool))  ->  (nat -> (IO client bool)).
intros f n; eapply IOget; apply f; assumption.Qed.
```

From the proof of this Lemma, using the *Recursive Extraction* command we obtain a Haskell program (together with all functions and datatypes involved):

```
rpc :: world -> world -> (Nat -> IO Bool) -> Nat -> IO Bool
rpc client server f n =  iOget client server (f n)
```

The type of the function extracted from the proof of `rpc` is essentially the same as of the specification, and its body is stripped of all logical parts. Notice that monadic constructors (e.g., `IOget`) are not unfolded in the extracted code, which therefore contains undefined symbols (`iOget`). We could define these symbols directly in Haskell, but this would be cumbersome and awkward due the distributed aspects we have to deal with. In this section, we discuss how this Haskell code is turned into a distributed Erlang program, by means of a "back-end compiler". (For an introduction to Erlang, we refer to [1,11].)

Let us denote by $(\!|\cdot|\!)_\rho$ the translating function from Haskell to Erlang, where ρ is the environment containing the Erlang module name and, for each identifier, a type telling whether this is a function or a variable, and in the former case the arity of that function. This is needed because in Haskell function variables begin with a capital letter, while in Erlang only variables do; the arity is needed to circumvent the currying of Haskell, which is not available in Erlang.

Most of Haskell syntax is translated into Erlang as expected, both being functional languages (but with different evaluation strategies). Monad operations need special care: each IO function must be implemented by a suitable code snippet, conforming to the intended meaning as given by their definition in Coq. One may think of Coq definitions as "pseudo-" or "high-level" code, and the Erlang implementations as the actual, "low level" code.

Implementation of Mobility: IOget. An application of the *IOget* constructor is extracted as a Haskell term (`iOget` A_1 A_2 $(F\ A_3)$), where A_1 is the actual world, A_2 is the remote world, and F is the term to be remotely evaluated with parameters A_3. This term is translated in Erlang as follows:

$$\begin{aligned}(\!|\texttt{iOget}\ A_1\ A_2\ (F\ A_3)|\!)_\rho = &\ \mathbf{spawn}\texttt{(element(2, }(\!|A_1|\!)_\rho\texttt{), }\rho\texttt{("modulename"),}\\ &\qquad\qquad \texttt{dispatcher,[fun () -> }(\!|F|\!)_\rho(\!|A_3|\!)_\rho\mathbf{end},\\ &\qquad\qquad (\!|A_2|\!)_\rho, \{\mathbf{self}\texttt{(), }\mathbf{node}\texttt{()}\}\texttt{]),}\\ &\ \mathbf{receive}\{\texttt{result, Z -> Z}\}\end{aligned}$$

Basically, this code **spawn**s a process on the remote host, sending it the code to be executed, and waits for the result with a synchronous **receive**. Let us examine the arguments of **spawn**. The first argument is the host address we are going to send the code to; in fact, in Erlang a process is identified by the pair `(pid, host-address)`, and hence we use these pairs as the implementations of λ_{XD} worlds. The second and third arguments are the module name and the function name to execute, respectively, which we will describe below. The fourth argument is a triple whose first component is the code to be executed applied to its parameters. Since Erlang is call-by-value, we have to suspend the evaluation of the program before sending to the remote site; to this end the application is packed in a vacuous λ-abstraction. The second and third components are the address of the final target (i.e., where the computation must be executed) and of the local node (i.e., the process which the result must be sent back to).

The function we spawn remotely is `dispatcher`, which takes a function and send it to another host where there is an `executer` function listening:

```
dispatcher(Mod, Fun, Executer, Target) ->
    spawn(element(2,Executer),update,executer,[Mod,Fun,Target]).
executer(Mod, Fun, FinalHost) ->
    Z = Fun(),  element(1,FinalHost) ! {result, Z}.
```

The dispatcher behaves as a code forwarder: it spawns a new executer process on the Executer machine passing it the code and the final target where the result has to be sent back. When the executer function receives the module, the function to execute and the target host, it simply evaluates the function by applying to (); then the result is sent back to the caller.

Implementation of References: IOnew, IOupdate, IOlookup. Being a declarative language, Erlang does not feature "imperative-style" variables and references. Nevertheless, we can represent a location as an infinite looping process which retains a value and replies to "read" messages by providing the value, and to "update" messages by re-starting with the new value:

```
location(X) -> receive
                 {update, Val} -> location(Val);
                 {get, Node} -> element(1, Node) ! {result, X}, location(X)
               end.
```

Therefore, creating a new location at a given world is simply spawning a new `location` process on the host corresponding to that world. The spawning primitive embedded in Erlang returns the pid of the spawned process which is passed to the continuation program. This is implemented in the translation of the `iOnew` function (which is extracted from the `IOnew` monad constructor):

$$(\!|\texttt{iOnew}\ A_1\ A_2\ A_3|\!)_\rho = ((\!|A_3|\!)_\rho)(\textbf{spawn}\texttt{(}\textbf{element}\texttt{(2, }(\!|A_1|\!)_\rho\texttt{)}, \rho\texttt{("module name"),}$$
$$\texttt{location, [}(\!|A_2|\!)_\rho\texttt{]))}$$

On the other hand, the implementation of `IOupdate` sends an "update" message to process whose pid is A_1, so it updates its internal value and continues with continuation program which takes no argument:

$$(\!|\texttt{iOupdate}\ w\ A_1\ A_2\ A_3|\!)_\rho = (\!|A_1|\!)_\rho!\{\textbf{update}, (\!|A_2|\!)_\rho\}, (\!|A_3|\!)_\rho$$

Notice that the Erlang typing discipline does not ensure either that A_1 is a pid referring to a local process (i.e., a local location), or that it is actually executing `location/1`. However, both these properties are guaranteed by Coq typing rules.

Finally the implementation of `IOlookup` is quite similar to `IOupdate`'s: the translation sends a "get" message, along with the caller address in order to get the response, and synchronously waits for the answer. The result of the operation is passed to the rest of the program:

$$(\!|\texttt{iOlookup}\ w\ A_1\ A_2|\!)_\rho = (\!|A_1|\!)_\rho\texttt{!}\{\texttt{get}, \{\textbf{self}(), \textbf{node}()\}\},$$
$$\textbf{receive}\{\texttt{result, Z}\} \rightarrow ((\!|A_2|\!)_\rho)(Z)$$

Example 2. Let us consider the remote procedure call function of Example 1. From that Haskell code, we can extract an Erlang procedure for the remote execution of any program that takes a natural, running our Haskell-to-Erlang compiler. We obtain the following Erlang program:

```
rpc (Client, Server, F, N)->
    spawn (element (2, Client), rpc, dispatcher, [rpc,
                          (fun () -> F (N)end),
                          Server,
                          { self (), node () } ]),
    receive { result, Z } -> Z end .
```

As expected, the extracted program has been translated into a spawn function which passes the term `X(H)` to the dispatcher on `w`, which in turn executes the function at `w'` by spawning a remote process; then it receives the result from the server, and finally returns it.

4 A Type Theory for Distributed Computations

The monadic type theory we have presented above can be used for specifying and synthesizing Erlang distributed programs, but it requires the user to have a non trivial knowledge of the Coq proof system. In fact, a programmer may prefer to use a high-level language, possibly focused on some particular aspects (e.g., web service orchestration, code mobility, etc.). Being more specialized, these *front* *end* languages may be not as expressive as the Coq internal logic, but simpler to use. Nevertheless, we can consider to translate these languages into CIC for taking advantage of the type checking and program extraction features of Coq.

As an example of this approach, in this section we present a simple type theory for distributed computation with references, called λ_{XD} ("lambda-cross-D"), and give it an interpretation in Coq using the monadic type theory above.

4.1 $\boldsymbol{\lambda_{XD}}$

The theory λ_{XD} is similar to Licata and Harper's HL5 [7], the main difference is that we have an explicit language for terms. This type theory has hybrid modal constructors for dealing with *worlds*, mobile computations, and references.

Syntax. The syntax of types and terms is defined as follows.

$$\begin{array}{lrl}
\textit{Types} & A ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{Ref} \\
 & & \mid A \times B \mid A \to B \mid \forall w.A \mid \exists w.A \mid A@w \mid \bigcirc A \\
\textit{Terms} & N, M ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \\
 & & \mid \mathsf{return}\ M \mid \mathsf{get\ w}\ M \mid \mathsf{bind}\ MN \mid \mathsf{assign}\ MN \mid \mathsf{new}\ M \\
 & & \mid \mathsf{deref}\ M \mid (M,N) \mid \pi_1\ M \mid \pi_2\ M \mid MN \mid \lambda x.M \\
 & & \mid \mathsf{hold}\ M \mid \mathsf{leta}\ x = M\ \mathsf{in}\ N \mid \mathsf{letd}\ (w,x) = M\ \mathsf{in}\ N \\
 & & \mid \mathsf{some}\ w\ M \mid \Lambda w.M \mid \mathsf{unbox}\ w\ M
\end{array}$$

The types $\to$ and $\times$ represent functions and products. The types $\forall$ and $\exists$ represent quantifiers over worlds. Next, @ is a connective of hybrid logic, which allows to be set the world at which a type must be interpreted. Finally, $\bigcirc$ and Ref represent monadic computations and references; note that they are not indexed by a world. The usual modal connectors $\Box$ and $\Diamond$ can be defined using quantifiers and @: $\Box A = \forall w.A@w$ and $\Diamond A = \exists w.A@w$.

Regarding terms, we have variables, numbers and booleans, etc. Constructors λ, leta, letd, Λ bind their variables as usual. leta is the usual "let" constructor, which binds x to the value of M, then evaluates N; letd is the elimination of existential types: M is expected to evaluate to some $u\ V$, and w, x are bound to u, V before evaluating N. Notice that worlds are not first class objects (this would lead to computations accepting and returning worlds, with a correspondingly more complex type theory). For monadic types we have the standard monadic constructors return and bind, plus constructors for *local* state manipulation such as deref, assign and new; the latter allocates a memory region and returns its address. Finally, get allows access to a remote resource as if it were local.

Typing Rules. Since the computations are related to worlds, a term can be given a type only with respect a world. Therefore, the typing judgment has the form "$\Gamma \vdash_{XD} t : A[w]$" which is read "in the context Γ, the term t has type A in the world w". (We will omit the index $_{XD}$ when clear from the context.) The typing contexts *Ctxt* are defined as usual: $\Gamma ::= \langle\rangle \mid \Gamma, x : A[w] \quad x \notin \mathrm{dom}(\Gamma)$

$$\frac{\Gamma, x : A[w] \vdash M : B[w]}{\Gamma \vdash \lambda x.M : A \to B[w]}\ \mathit{Lam} \qquad \frac{\Gamma \vdash M : A \to B[w] \quad \Gamma \vdash N : A[w]}{\Gamma \vdash (M\ N) :\ B[w]}\ \mathit{App}$$

$$\frac{\Gamma \vdash M : A[w] \quad \Gamma \vdash N : B[w]}{\Gamma \vdash (M, N) : A \times B[w]}\ \mathit{Pair} \quad \frac{\Gamma \vdash M : A \times B[w]}{\Gamma \vdash \pi_1 M : A[w]}\ \mathit{Proj}_1 \quad \frac{\Gamma \vdash M : A \times B[w]}{\Gamma \vdash \pi_2 M : B[w]}\ \mathit{Proj}_2$$

$$\frac{\Gamma \vdash M :\ A[w'] \quad w \text{ fresh in } \Gamma}{\Gamma \vdash \Lambda w.M : \forall w.A[w']}\ \mathit{Box} \qquad \frac{\Gamma \vdash M : \forall w.A[w]}{\Gamma \vdash \mathsf{unbox}\ w' M : A[w]}\ \mathit{Unbox}$$

$$\frac{\vdash \Gamma \quad x : A[w] \in \Gamma}{\Gamma \vdash x : A[w]}\ \mathit{Var} \qquad \frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \mathsf{some}\ w\ M : \exists w.A[w]}\ \mathit{Some}$$

$$\frac{\Gamma \vdash M : \exists u.A[w] \quad \Gamma, x : A\{z/u\}[w] \vdash N : C[w'] \quad z \text{ fresh in } \Gamma}{\Gamma \vdash \mathsf{letd}\ (z, x) = M\ \mathsf{in}\ N : C[w']}\ \mathit{LetD}$$

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \mathsf{hold}\ M : A@w[w']}\ \mathit{Hold} \qquad \frac{\Gamma \vdash M : A@z[w] \quad \Gamma, x : A[z] \vdash N : C[w]}{\Gamma \vdash \mathsf{leta}\ x = M\ \mathsf{in}\ N : C[w]}\ \mathit{LetA}$$

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \mathsf{return}\ M : \bigcirc A[w]}\ \mathit{Ret} \qquad \frac{\Gamma \vdash M : \bigcirc A[w] \quad \Gamma \vdash N : A \to \bigcirc B[w]}{\Gamma \vdash \mathsf{bind}\ M\ N : \bigcirc B[w]}\ \mathit{Bind}$$

$$\frac{\mathsf{Mobile}\ A \quad \Gamma \vdash M : \bigcirc A[w']}{\Gamma \vdash \mathsf{get}\ w'\ M : \bigcirc A[w]}\ \mathit{Get} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}[w] \quad \Gamma \vdash N : \mathsf{Ref} \to \bigcirc A[w]}{\Gamma \vdash \mathsf{new}\ M\ N : \bigcirc A[w]}\ \mathit{New}$$

$$\frac{\Gamma \vdash M : \mathsf{Ref}\ [w] \quad \Gamma \vdash N : \mathsf{Nat} \to \bigcirc A}{\Gamma \vdash \mathsf{lookup}\ M\ N : \bigcirc A[w]}\ \mathit{Lookup}$$

$$\frac{\Gamma \vdash T1 : \mathsf{Ref}\ [w] \quad \Gamma \vdash T2 : \mathsf{Nat}[w] \quad \Gamma \vdash T3 : \bigcirc A[w]}{\Gamma \vdash \mathsf{update}\ T1\ T2\ T3 : \bigcirc A[w]}\ \mathit{Update}$$

Fig. 1. Typing system for λ_{XD}

The typing system is given in Figure 1. Many typing rules are straightforward; we just focus on the most peculiar ones.

For monadic types, beside the standard rules for return and bind, we have rules for references and mobile computation. We have chosen that references can contain only naturals (rules *New, Update, Lookup*), but clearly this can be easily generalized. A computation at a given world can be "moved" to another only by means of the get primitive, which can be seen as a "remote procedure call". However, not all computation types can be moved safely; in particular, a reference is meaningful only with respect to the world it was generated from. Therefore, following [7], we define the class of *mobile types*. Intuitively, a type is mobile if its meaning does not depend on the world where it is interpreted. This is formalized by an auxiliary judgment Mobile over types, defined as follows:

$$\frac{b \in \{\mathsf{Unit},\ \mathsf{Nat},\ \mathsf{Bool}\}}{\mathsf{Mobile}\ b} \qquad \frac{}{\mathsf{Mobile}\ A@w} \qquad \frac{\mathsf{Mobile}\ A \quad \mathsf{Mobile}\ B}{\mathsf{Mobile}\ A \times B} \qquad \frac{\mathsf{Mobile}\ A\ \ Q \in \{\forall, \exists\}}{\mathsf{Mobile}\ Qw.A}$$

A computation returning Nat can be accessed remotely, even if it has some side effect on the remote world, like e.g., $\mathsf{get}\ w'\ ((\mathsf{new}\ 0\ (\lambda x.\mathsf{lookup}\ x\ (\lambda y.\mathsf{return}\ y)))$ (which has type $\bigcirc\mathsf{Nat}$ at any world w). On the other hand, a computation of type $\bigcirc\mathsf{Ref}$ cannot be executed remotely. Still, a reference becomes mobile if we indicate which world it comes from, using the hold constructor; e.g., the term $\mathsf{get}\ w'\ ((\mathsf{new}\ 0(\lambda x.\mathsf{return}\ (\mathsf{hold}\ x)))$ has type $\bigcirc(\mathsf{Ref}@w')$ at any world w.

4.2 Formalization of λ_{XD} in Coq

The next step is to formalize the object type theory in the Calculus of Inductive Constructions, by means of a "deep encoding" of types and terms.

Types are represented by an inductively defined `Set`. The definition is standard, using the "weak HOAS" technique for representing binders $\forall$ and $\exists$ [3,2]:

```
Hypothesis world : Set.                | typ_fun: typ -> typ -> typ
Inductive typ : Set :=                 | typ_forall : (world -> typ) -> typ
  typ_unit : typ                       | typ_exists : (world -> typ) -> typ
| typ_nat : typ                        | typ_at : typ -> world -> typ
| typ_bool : typ                       | typ_monad : typ -> typ
| typ_pair : typ -> typ -> typ         | typ_ref : typ.
```

As an example, $\forall w.A@w$ is represented as `(typ_forall (fun w => (typ_at A w))`. Notice that `world` is not defined inductively (otherwise we would get exotic terms), the only terms inhabiting `world` are variables and possibly constants.

The encoding of terms deals with two kinds of variables and binders. Since we will define a typing judgment with explicit typing contexts, we use de Bruijn indexes for representing first-class variables x; instead, variables w ranging over worlds are represented via weak HOAS, as shown in the following definition:

```
Inductive location : Set := loc : world -> nat -> location.
Inductive term : Set :=
  term_var : nat -> term                | term_pair : term -> term -> term
| term_tt : term                        | term_prj1 : term -> term
| term_location : location -> term      | term_prj2 : term -> term
| term_o : term                         | term_app : term -> term -> term
| term_s : term -> term                 | term_lam : term -> term
| term_return : term -> term            | term_hold : term -> term
| term_get : world -> term -> term      | term_leta : term -> term -> term
| term_bind : term -> term -> term      | term_letd : term -> (world -> term)
| term_new : term -> term -> term                    -> term
| term_update : term -> term ->         | term_some : world -> term -> term
                term -> term            | term_box : (world -> term) -> term
| term_lookup : term -> term -> term | term_unbox : world -> term -> term.
```

Thus, `(typ_fun (typ_var 0))` is the usual "de Bruijn-style" encoding of $\lambda x.x$, while `(typ_box fun w => (term_get w M))` is the encoding of $\Lambda w.(\mathsf{get}\ w\ 0)$ in "weak HOAS style". The constructor letd is represented by `term_letd` using both techniques at once: weak HOAS for the world binding and de Bruijn indexes for the term binding. This mixed approach has many advantages: first, we do not need an additional type to represent term variables, secondly, the correctness is easier to prove. Other language constructors are self-explaining.

Finally the typing judgment $\vdash_{XD}$ is represented by an inductive type as follows (for lack of space, we show only the most complex rule, i.e., that for letd)

```
Inductive typing: env -> term -> typ -> world -> Set := ...
| typing_letd: forall E t1 t2 A C w w',
  E |= t1 ~: typ_exists A [ w' ] ->
    (forall z, E & A z ~ w' |= t2 z ~: C [ w ]) ->
    E |= term_letd t1 t2 ~: C [ w ]
```

where `env` is the datatype of typing contexts, in de Bruijn style:

```
Inductive env:Set := env_empty:env | env_cons:env -> typ -> world -> env.
```

Intuitively, the term `E |= t ~: A [ w ]` represents the typing judgment $\Gamma \vdash_{XD} M : A[w]$. More formally, we can define the obvious encoding functions $\epsilon_{Terms} : \mathit{Terms} \to$ `term`, $\epsilon_{Types} : \mathit{Types} \to$ `typ`, $\epsilon_{Ctxt} : \mathit{Ctxt} \to$ `env`, such that the following holds (where we omit indexes for readability):

Proposition 1. *For all type A, term M, world w and context Γ, if the worlds appearing free in A, M, Γ, w are $w_1, \dots, w_n$, then $\Gamma \vdash_{XD} M : A[w] \iff$* `w1:world,...,wn:world` $\vdash$ `_:` $\epsilon(\Gamma)$ `|=` $\epsilon(M)$`~:` $\epsilon(A)$ `[ w ]`.

This result can be proved by induction on the derivation of $\Gamma \vdash_{XD} M : A[w]$ ($\Rightarrow$) and on the derivation of $\epsilon(\Gamma)$`|=`$\epsilon(M)$ `~:` $\epsilon(A)$ `[ w ]` ($\Leftarrow$). In virtue of this result, in the following we will use the "mathematical" notation for types, terms and typing judgments (i.e., $\Gamma \vdash M : A[w]$) in place of their Coq counterpart.

4.3 Translation of λ_{XD} into CIC Sets

In this section we give the translation of types and well-typed terms of the object type theory λ_{XD}, into sets and terms of the Calculus of Inductive Construction, respectively, using the IO monad. This translation can be seen as a way for providing a *shallow* encoding of λ_{XD} in Coq.

Interpretation of Object Types. Object types are interpreted in worlds by a function $\cdot$`<·>` : `typ -> world -> Set`, inductively defined on its first argument:

$$\begin{aligned} \mathsf{Unit}\,\texttt{<w>} &= \texttt{unit} & A \times B\,\texttt{<w>} &= A\,\texttt{<w>}\ \texttt{*}\ B\,\texttt{<w>} \\ \mathsf{Nat}\,\texttt{<w>} &= \texttt{nat} & A \to B\,\texttt{<w>} &= A\,\texttt{<w>}\ \texttt{->}\ B\,\texttt{<w>} \\ \mathsf{Bool}\,\texttt{<w>} &= \texttt{bool} & \forall w'.A\,\texttt{<w>} &= \texttt{forall w'}, (A\ w')\,\texttt{<w>} \\ \mathsf{Ref}\,\texttt{<w>} &= \texttt{ref}\ w & \exists w'.A\,\texttt{<w>} &= \{\ \texttt{w' : world}\ \&\ (A\ w')\,\texttt{<w>}\ \} \\ A@w'\,\texttt{<w>} &= A\,\texttt{<w'>} & \bigcirc A\,\texttt{<w>} &= \texttt{IO w}\ (A\,\texttt{<w>}) \end{aligned}$$

Basic types are translated into native Coq types, except for references which are mapped to a specific data type, as explained below. Most constructors are interpreted straightforwardly as well. In particular, note that $\exists w'.A$ is interpreted as a Σ-type, whose elements are pairs `(z,M)` such that `M` has type $(A\ z)$`<w>`. The type $A@w'$ is translated simply by changing the world in which A is interpreted. Finally, the lax type $\bigcirc A$ is translated using the type monad `IO w : Set -> Set`, parametric over worlds, defined in Section 2.

Now we can extend the intepretation of types to typing judgments. Loosely speaking, a judgment $\Gamma \vdash t : A[w]$ should be interpreted as a term of type $\Gamma \to A[w]$. More precisely, this `Set` is obtained by a Coq function $[\![\ ]\!]$: `env -> typ -> world -> Set`, defined recursively over the first argument, such that:

$$[\![(A_1[w_1], ..., A[w_n]), A[w]]\!] = A_1\,\texttt{<w}_1\texttt{>}\ \texttt{*}\ \ldots\ \texttt{*}\ A_n\,\texttt{<w}_n\texttt{>}\ \texttt{->} A\,\texttt{<w>}$$

Interpretation of Object Terms. Once the type theory has been interpreted as `Set`s, we can give the interpretation of λ_{XD} terms as functions among these sets. Due to lack of space we do not describe in detail this translation, as it is as expected. Most constructors are immediately mapped to their semantic counterparts, e.g., pair is mapped to `pair`, some to `exist`, λ to abstraction, etc. Monadic constructors like `term_return`, `term_get`, ..., are mapped to the corresponding constructors of the IO monad. For more details, we refer to the Coq code.

Soundness. Now, we aim to prove that if a λ_{XD} type A is inhabited by a term t, then also the interpretation of A must be inhabited—and the corresponding inhabitant is obtained by translating the object term t. Before stating and proving this result, we need a technical lemma about mobile types:

Lemma 1 (Mobility). *For all $A \in Type$, if Mobile A, then for all w, $w' \in$ `World`, A <w> $=$ A <w'>.* `[Coq proof]`

This result means that "mobile" types can be translated from one world to another. The Mobile assumption is needed because there are types whose interpretation cannot be moved from one world to another (e.g., Ref). However this "remote access" property is needed only in the case of the get constructor, and its typing rule requires the type to be mobile (Figure 1), so Lemma 1 applies.

Theorem 1 (Soundness). *Let $\Gamma \in Ctxt$, $t \in Term$, $A \in Type$, $w \in World$, let $\{w_1, \ldots, w_n\}$ be all free worlds appearing in Γ, A, t, w. Then, if $\Gamma \vdash_{XD} t : A[w]$ then there exists a term $[\![t]\!]$ such that* `w1:world,...,w2:world` $\vdash [\![t]\!] : [\![\Gamma, A[w]]\!]$. `[Coq proof]`

Proof. (sketch) The proof is by induction on the derivation of $\Gamma \vdash_{XD} t : A[w]$. Most cases are easy; in particular, constructors of monadic types are translated using the constructors defined above.

Let us focus on the case of get, where we move from $\bigcirc A[w']$ to $\bigcirc A[w]$ with the rule *Get*. In the translation we have to build a term of type $\bigcirc A$ `<w>`, i.e., `IO w (`A`<w>)`, from a term in `IO w' (`A`<w'>)` given by inductive hypothesis. In fact, this type is equal to `IO w (`A`<w'>)`, because A is Mobile and by Lemma 1. Then, using `IOget` we can replace `w'` with `w` as required.

For the remaining cases, we refer the reader to the Coq script. □

It is worth noticing that this theorem is stated in Coq as follows

```
Theorem soundness:  forall (t : term),
    forall (w: world), forall (A : typ), forall (E : env),
      E |= t ~: A [ w ] -> Interp E A w.
```

and its proof is precisely the encoding function from λ_{XD} well-typed terms to their semantic interpretations in Coq, i.e., functions among Sets.

Remark 2. In this section, we have encoded separately the type of terms and the typing judgment. Another possibility is to define an inductive type of *implicitly typed* terms, representing both the syntax and the typing system, as in [7]:

```
Inductive term : env -> typ -> world -> Set :=
  term_var : nat -> term
| term_o : forall G w, (term G typ_nat w)
| term_s: forall G w, (term G typ_nat w) -> (term G typ_nat w)
| term_fun : forall A B G w, (term G::(A,w) B w) ->
                         (term G (typ_fun A B) w)
| term_app : forall A B G w, (term G (typ_fun A B) w) ->
                         (term G A w) -> (term G B w)
| ...
```

In this way, terms inhabiting (term G A w) are automatically well typed. Also the soundness statement would be simplified accordingly:

```
Theorem soundness: forall G A w, (term G A w) -> (Interp G A w).
```

Although this approach may simplify a bit the technical development in Coq, we have preferred to keep separated terms and type system because it is sticking to the original definition "on the paper" (i.e., that in Section 4.1), and hence easier to adapt to other languages and type systems.

5 Example: Synthesis of Remote Read/Write

In this section we describe an example application of our framework, showing how to give the specification of two distributed functions which have to satisfy together some property. In particular, we want to show that, if we store a given value in some remote location created on the fly and afterwards we read from the same address, we will find exactly the same value and no error can arise. From the proof of this property, we will extract the corresponding Erlang code, yielding two functions, remoteread and remotewrite, which behave as required.

We begin with giving the type of the remotewrite function which takes the value at the world w1 and gives a computation at w1 which produces a location at the world w2.

```
Lemma remotewrite: forall w1 w2:world,
            (typ_nat @ w1 ==> (O (typ_ref@w2))) < w1 >.
```

Notice that this specification does not guarantee that the function will not run without errors, since the monad allows also for faulty computations.

The remoteread function takes the reference to the location and gives a computation which returns the value contained in the location. Notice the world of the location is different from the world of the resulting computation:

```
Lemma remoteread: forall w1 w2, typ_ref@w2 ==> (O typ_nat) < w1 >.
```

Clearly, these two specifications do not impose anything about the combined behaviour of these two functions—they just declare their types. In order to achieve the correct specification, we have to state and prove the following lemma, declaring that there exists a computation and a function which behave correctly:

```
Lemma update: forall (w w': world) (value : typ_nat < w >),
    {o : (O (typ_ref@w') < w >) *
        ((typ_ref@w') ==> O typ_nat < w >) |
      forall s, getvalue ((IObind (fst o) (fun l => (snd o) l)) s)
                 = Some value}.
```

This lemma can be proved using the properties of the monad operators. Then, executing Coq's `Extraction` command, we obtain an Haskell code, which can be translated to Erlang by our back-end compiler. In the end, we obtain the following distributed code:

```
remotewrite (W1, W2, Value)->
  spawn (element (2, W1),
         update2,
         dispatcher,
         [update2,
          (fun () -> (fun (Address)-> Address end)
                     (spawn (element (2, W2),
                     update2,
                     location, [Value])) end),
          W2, {self (), node ()}]),
  receive {result, Z} -> Z end .
remoteread (W1, W2, Address)->
  spawn (element (2, W1),
         update2,
         dispatcher,
         [update2,
          (fun () -> Address ! {get, {self (), node ()}},
                     receive {result, X0} ->
                        (fun (H)-> H end)(X0)
                     end end),
          W2, {self (), node ()}]),
  receive {result, Z} -> Z end.
```

The function `remotewrite` spawns a process at `W1` with final destination `W2`, which creates a new location with `Value` as initial value and waits for the address of that location. On the other hand, `remoteread` spawns a process to `W1` with final target `W2`, which sends to the `Address` location a request for the value and waits for the response. The target process computes and sends back the result of the computation, which is received by the final `receive`.

Clearly, these functions can execute only when they are called by some other program. we can define the `main/0` function, which can be seen as the "orchestration" part for executing the distributed code. As an example, let us consider the following implementation where we declare the remote world `Pub` and the

`Dispatcher`, then `remotewrite` will create a new location and put in it the value 254. `remoteread` is then called to read the same value from the same address:

```
main() -> Pub = {pub, 'pub@<IP>'},
          Dispatcher = {disp, 'disp@<IP>'},
          Val = 254,
          Address = remotewrite(Dispatcher, Pub, Val),
          Value = remoteread(Dispatcher, Pub, Address).
```

6 Conclusions

In this work we have presented a methodology for the synthesis of distributed, mobile programs in Erlang, through the extraction facility offered by Coq. First, we have defined monadic types in Coq, covering the computational features we are dealing with (i.e., side-effects and distributed computation). These monadic types can be used in the specification of Haskell programs, which can be obtained by extraction from the proofs of these specifications. These programs contain monadic constructors for distributed imperative computations, which are exploited by a "post-extraction" Haskell-to-Erlang compiler, which generates the requested distributed mobile program. Moreover, in order to simplify the burden of programming in Coq, we have defined λ_{XD}, a monadic type theory for distributed computations similar to Licata and Harper's HL5, which can be seen as a *front-end* programming language. In fact, this type theory has been given a formal interpretation within the Calculus of Inductive Constructions, which allows a λ_{XD} type to be converted into a CIC specification and a λ_{XD} program into a CIC proof. Using the back-end compiler above, these proofs can be turned into runnable Erlang programs.

Several directions for future work stem from the present one. First, we can extend the language to consider also worlds as first-class objects. This would allow computations to take and returns "worlds", so that a program can dynamically choose the world where the execution should be performed.

Although we have considered distributed computations with references, our approach can be ported to other computational aspects, possibly implemented in other target languages. The monad `IO` has to be extended with new (or different) constructors (possibly in a quite different target language), the post-extraction back-end compiler should be extended to cover these new constructors, and the front-end type theory must be adapted as needed.

However, the most important future work is to prove that the post-extraction compiler, i.e., the translation from Haskell to Erlang, is correct. To this end, we could follow the approach of the CompCert project described by Leroy [4,5]. This can be achieved by giving in Coq a formal semantics to the fragments of Haskell and Erlang that the back-end compiler targets. In particular, the crux of the correctness proof is proving that the Erlang implementations of mobility, plus state effects as threads, are correct with respect to the "reference implementation" of the corresponding monad operators.

Acknowledgments. The authors are grateful to the anonymous reviewers for many useful and important remarks, which allowed to quite improve the paper.

References

1. Armstrong, J.: Erlang - a survey of the language and its industrial applications. In: Proc. INAP 1996 (1996)
2. Honsell, F., Miculan, M.: A Natural Deduction Approach to Dynamic Logics. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 165–182. Springer, Heidelberg (1996)
3. Honsell, F., Miculan, M., Scagnetto, I.: π-calculus in (co)inductive type theory. Theoretical Computer Science 253(2), 239–285 (2001)
4. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Proc. POPL, pp. 42–54. ACM (2006)
5. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
6. 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)
7. Licata, D.R., Harper, R.: A monadic formalization of ML5. In: Crary, K., Miculan, M. (eds.) Proc. LFMTP. EPTCS, vol. 34, pp. 69–83 (2010)
8. Murphy VII, T., Crary, K., Harper, R.: Type-Safe Distributed Programming with ML5. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 108–123. Springer, Heidelberg (2008)
9. Paulin-Mohring, C., Werner, B.: Synthesis of ML programs in the system COQ. Journal of Symbolic Computation 15, 607–640 (1993)
10. Plotkin, G., Power, J.: Notions of Computation Determine Monads. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002)
11. Virding, R., Wikström, C., Williams, M.: Concurrent programming in ERLANG, 2nd edn. Prentice Hall International (UK) Ltd. (1996)

A Erlang Concurrency Fragment

A process is a self-contained, separate unit of computation which exists concurrently with other processes in the system. There is no inherent hierarchy among processes; the designer of an application may explicitly create such a hierarchy.

The `spawn`/3 function creates and starts the execution of a new process:

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

`spawn`/3 creates a new concurrent process to evaluate the function and returns its `Pid` (process identifier). The call to `spawn`/3 returns immediately when the new process has been created and does not wait for the given function to evaluate. We want point out that Erlang does not distinguish between a local spawn or a remote spawn, in the sense that *the pid the spawn returns is unique and global for all the hosts* and we can use it transparently. So, a spawn like `Pid=spawn(Host, Module, Function, ArgumentList)`, will create a *remote* process on the `Host` target and the `Pid` can be used to communicate with it. The process identifier of the new process is now known only to process which called the spawn function. Pids are necessary for all forms of communication, so *security in an Erlang system is based on restricting the spread of the Pid of a process.*

In Erlang, processes can communicate only by message passing, according to the "actors" model. A message is sent to a process by the primitive '`!`' (send):

```
Pid ! Message
```

`Pid` is the identifier of the process to which `Message` is sent. A message can be any valid Erlang term, which is evaluated before being sent; then, `send` returns the message sent as its value. Sending a message is an asynchronous operation so the send call will not wait for the message either to arrive at the destination or to be received. This is in line with the asynchronous nature of message passing: the application must itself implement all forms of checking. Messages are always delivered to the recipient, and in the same order they were sent. Moreover, since Erlang is a full-fledged functional language, messages can be also functions.

The primitive `receive` is used to receive messages:

```
receive
    Message1 -> Actions1 ;
    Message2 -> Actions2 ;
    ...
end
```

Each process has a mailbox and all messages which are sent to the process are stored in the mailbox in the same order as they arrive. In the above, `Message1` and `Message2` are patterns which are matched against messages that are in the process's mailbox. When a matching message is found and any corresponding guard succeeds the message is selected and removed from the mailbox. `receive` acts as any other forms of pattern matching: any unbound variabl in the message pattern becomes bound. `receive` is blocking, i.e., the process evaluating it will be suspended until a message is matched.

Towards Provably Robust Watermarking*

David Baelde[1,2], Pierre Courtieu[3], David Gross-Amblard[4],
and Christine Paulin-Mohring[1]

[1] LRI-PROVAL – Université Paris Sud/CNRS/INRIA, Saclay, France
[2] IT University of Copenhagen, Denmark
[3] CÉDRIC – CNAM, Paris, France
[4] IRISA – Université Rennes 1, Rennes, France

Abstract. Watermarking techniques are used to help identify copies of publicly released information. They consist in applying a slight and secret modification to the data before its release, in a way that should be *robust*, *i.e.*, remain recognizable even in (reasonably) modified copies of the data. In this paper, we present new results about the robustness of watermarking schemes against arbitrary attackers, and the formalization of those results in Coq. We used the Alea library, which formalizes probability theory and models probabilistic programs using a simple monadic translation. This work illustrates the strengths and particularities of the induced style of reasoning about probabilistic programs. Our technique for proving robustness is adapted from methods commonly used for cryptographic protocols, and we discuss its relevance to the field of watermarking.

1 Introduction

Watermarking consists in embedding some information inside a document in a *robust* and usually *imperceptible* way. It is notably used in digital property claims: a content owner may mark a document before its release, in order to be able to recognise copies of it and claim ownership. Of course, this cannot be done in a reliable and convincing way unless properties of the watermarking scheme have been solidly established.

A wide literature is dedicated to techniques for marking sound, still images and videos [7] that are robust against common transformations such as scaling, resampling, cropping, etc. This is achieved by using meaningful notions of distance and performing watermarking in an adequate space, such as frequency spectra. One may also consider documents as unstructured bit strings, and measure distortions using the Hamming distance, *i.e.*, the number of positions where bits differ. In this setting, simple watermarking techniques may be used. The *bit-flipping* scheme involves a secret mask K which is a bitstring of the same length as the original document that is to be marked. Marking a document O (called *support* in this context) is done by changing the value of bits at positions set in K

* Work partially supported by SCALP project ANR-07-SESU-010 and grant 10-092309 from the Danish Council for Strategic Research.

— we say a position is set when the corresponding bit is 1. This can be expressed as a bitwise xor: $\texttt{mark}(K, O) = O \oplus K$. The induced distortion is the number k of bits set in K; this quantity is typically chosen to be a small fraction of the size of the support, and its value is assumed to be public. Detection of suspect copies O', written $\texttt{detect}(O, K, O')$, is done by counting the number of positions set in K where O and O' differ — in other words, the number of positions set in K where O' coincides with $\texttt{mark}(K, O)$. If that number exceeds a predefined threshold, the suspect document O' is claimed to be a copy of $\texttt{mark}(K, O)$. The idea behind this scheme is that an attacker who does not know K has little chance of changing enough bits for evading detection unless it also distorts the document so much that it renders it worthless. A variant of this technique is *substitution* watermarking where, instead of flipping bits at chosen positions in the original support, a secret message is used as a substitution for those bits. It is important to point out that although working with unstructured bitstrings is idealized, it has practical counterparts. For instance, the Agrawal, Haas and Kiernan's method [2] for watermarking numerical databases hides a mark by applying a substitution to least significant bits, and quality measurement is proportional to the Hamming distance.

Watermarked documents may be subject to attacks by malevolent users. A protocol is said to be *robust* against a particular attack if the mark can still be detected with high probability in copies modified by this attack. As observed above, it is useless to consider attacks that introduce excessive distortion: though such attacks are usually easy, they result in valueless data. Numerous attack scenarios can be envisioned, and it is impossible to test them all one by one. Nevertheless, we expect a protocol to be *provably robust*, *i.*e., robust against any attack. Provable security for watermarking is difficult to achieve, and there is currently little work in that direction. One reason for this is that watermarking protocols may be attacked at various levels, sometimes independently of the nature of documents and the marking algorithms. For example, in the context of digital property, someone may apply his own mark to marked data in order to claim ownership. To address such issues separately, Hopper, Molnar and Wagner [9] have introduced a distinction between strong watermarking schemes and non-removable embeddings (NREs). The former refers to the general protocol used to claim ownership, while the latter is the core technique for marking documents, consisting of a marking and a detection algorithm. Using cryptography and trusted third parties, they have formally proved (on paper) that strong watermarking protocols, robust against arbitrary attackers, can be derived from NREs. However, they did not address the robustness of particular embedding techniques, which depends on the kind of document that is considered. More generally, there is surprisingly little literature on robustness against arbitrary attackers, even in the idealised setting of bitstring algorithms. The problem has been identified, and several authors have proposed definitions and methodologies [1,11], but they did not provide proofs of their results even for simple schemes. To the best of our knowledge, the only such proof has been carried out in the context of graph watermarking [12]. This proof relies on complex assumptions, concerning in particular the limited knowledge of the attacker, which may be hard to formalise. In contrast, our approach is very elementary, relying on a clear formalisation and basic combinatorics. We also note that, although a central piece in that proof of robustness for graphs is the (proved) robustness of a technique for marking integer vectors, this technique is not suitable for marking bitstrings under the Hamming distance, since it applies a modification to each component of the vector — which also eases considerably the robustness argument.

In this paper, we make two important steps towards provably secure watermarking protocols. First, we present a new proof of robustness against arbitrary attackers, which applies to bit-flipping and substitution watermarking for bitstrings. The key idea here is an efficient reduction of robustness to secrecy of the key K, which is used to obtain a tight enough bound on the probability of successful attacks. Second, we have fully formalised these proofs using the Coq proof assistant and the Alea library for reasoning about probabilities and probabilistic programs. Therefore, we provide a solid methodology for further work in the area. The formalisation of our work is also interesting in that it involves aspects such as information, secret and probabilistic algorithms which are challenging for the formal methods community. There are indeed few significant formalisations of randomised programs, although this field is receiving increasing attention [3,8,10]. Our work is most closely related to Certicrypt [4,5], a framework built on top of Alea for formalising proofs of cryptographic protocols. Those proofs proceed by program transformations, reducing cryptographic games (represented using a deep embeding approach) to simpler ones corresponding to mathematical assumptions on cryptographic primitives, *e*.g., the computational hardness of discrete logarithms in cyclic groups. As we shall see, our work on watermarking does not necessitate a model as complex as the one used in Certicrypt. But the main difference lies in the nature of our proofs: while our argument also revolves around a reduction, it mainly relies on concrete computations on bitstrings. The end result is also quite different: our proof is self-contained and does not use assumptions about complex mathematical primitives. Therefore, our Coq development provides an interesting case study for Alea, showing how elements of computational information theory may be carried out in that framework.

The rest of the paper is structured as follows. In Section 2 we motivate high-level modelling choices and present the Alea library which is used to formally realise these choices in Coq. Then, we describe our robustness proof in Section 3, detailing the main steps of the reduction for bit-flipping watermarking, and showing how it applies to substitution watermarking as well. We conclude by discussing our results and directions for future work in Section 4.

2 Formalisation

Security properties are commonly expressed as games played by the implementation of a service against an attacker. This methodology has the advantage of providing a clear and concrete characterisation of the flaw we want to avoid. The security of NREs has been defined in this way [9], and we shall formally define robustness as a variant of it. We assume that $\texttt{dist}(O, O')$ represents the distance between O and O' and that $\texttt{detect}(O, K, O')$ is true when O' is a marked copy of O with key K. An attacker $\mathcal{A}$ will break the robustness of a watermarking scheme when the following game answers true, *i.e.*, the attacker produced an object close to the original and the mark is no longer detected:

$$\begin{array}{l} O' \leftarrow \texttt{mark}(K, O) \\ O'' \leftarrow \mathcal{A}(O') \\ \texttt{dist}(O', O'') < \gamma \wedge \neg\texttt{detect}(O, K, O'') \end{array}$$

Of course, attacking watermarking schemes in this game is usually going to be easy unless we assume that the attacker does not know K and O. Other parameters affect the difficulty and meaning of our game, most notably the degree of abstraction of the model and the assumptions on the computational resources of the attacker. These parameters have to be investigated in order to obtain a meaningful formalisation.

Symbolic models, where secrets are perfect and cannot be guessed, can be used to reveal security breaches in some protocols. However, their high level of abstraction would trivialise the study of watermarking schemes. We are interested in quantifying how hard it is for an attacker to remove a mark, potentially relying on random choices and leaked information. In order to do this, we place ourselves in a probabilistic computational model where secrets are regular data that has been randomly chosen and may therefore be guessed by another party. In that context, our robustness game can be rephrased as follows, where all computations are probabilistic:

$$\begin{array}{l} O \leftarrow \texttt{random\_support}() \\ K \leftarrow \texttt{random\_key}() \\ O' \leftarrow \texttt{mark}(K, O) \\ O'' \leftarrow \mathcal{A}(O') \\ \texttt{dist}(O', O'') < \gamma \wedge \neg\texttt{detect}(O, K, O'') \end{array}$$

The computational resources available to the attacker are also a very important parameter of security games. In the case of cryptography, security often relies on the (supposed) hardness of a few problems: for example, inverting a logarithm is essentially as hard as guessing its argument. Under such circumstances, it is possible for an attacker to succeed if it is allowed to compute long enough, but security can be preserved if the time needed is exponential in a parameter that can be chosen at will, typically the size of the secret key. The case of watermarking differs a lot. In that context, the size of the key is limited by the support, and does not offer significant control on the computational cost of an attack. But it does not matter because this cost is in fact irrelevant: unlike in cryptography, the attacker has (usually) no way to know if he succeeded, which makes brute force attacks impossible. Therefore, we only need to prove that the attacker has a low probability of removing the mark in one attempt, and we can do so without any constraint on the time or space available to the attacker.

As we have seen, formalising robustness requires the notions of probability and probabilistic program. Since we do not need to make restrictions on the use of resources by attackers, they can also be modelled using the same notion of probabilistic algorithm. Thus, we chose to carry out our formalisation in the Coq library Alea, since it provides exactly those fundamental ingredients. Note, however, that there is no essential reason why our work could not be done with another proof assistant.

2.1 Alea

Alea is a Coq library containing a formalisation of complete partial orders, of the unit interval `U` ($[0;1]$), of probability theory, and tools for reasoning about probabilistic programs. We quickly recall some of the key concepts used in Alea. More details can be found in [3].

A standard, deterministic program of type A evaluates to a single value of type A. A probabilistic program may evaluate to different values in different runs, defining a probability distribution over A. This is the approach taken in Alea, which provides machinery for building such distributions in a functional programming style, using a monadic interpretation.

A probabilistic program of type A is thus viewed as a distribution, *i*.e., a Coq object of type `distr` A. The distribution may also be called a measure, or experiment. It is something that one can integrate over: it roughly has the type $(A \to [0;1]) \to [0;1]$, taking a weight function over A and returning its sum over the measure. The integral $\int f\ dP$ is written `mu P f` in Coq (we also use the notation $\mu\ P\ f$). Given a property Q on A, let 1_Q be the function which is 1 when Q holds and 0 otherwise, then $\mu\ P\ 1_Q$ represents the probability that the output of program P satisfies Q. The distribution P comes with some properties, *e*.g., it must be guaranteed to preserve addition, $\mu\ P\ (f+g) = (\mu\ P\ f)+(\mu\ P\ g)$.

The basic programming constructs for randomised computation include:

– **return** a which given $a : A$, returns the Dirac's distribution at point a;
– **let** $x = d_1$ **in** d_2 which given a probabilistic program d_1 of type `distr` A and a probabilistic program d_2 of type `distr` B depending on x of type A, evaluates d_1, bind the resulting value to x and then evaluates d_2.

As an example of Alea, we shall show how to encode probabilistic Turing machines. We need a set Σ of states and a set A of letters. A predicate `accept` of type $\Sigma \to$ `bool` distinguishes accepting states. The output of a transition will be a record $\{$`next` $: \Sigma;$ `write` $: A;$ `move` $:$ `dir`$\}$ with `dir` the type of directions with two values `L` and `R`. A transition is a function `trans` which takes a state and a letter as arguments and gives an output. The configuration of the machine is given by a state, a tape and the current position of the tape, it is represented by a record $\{$`cur` $: \Sigma;$ `tape` $:$ `Z` $\to$ `A`$;$ `pos` $:$ `Z`$\}$.

It is trivial to define a deterministic function `update` which given a configuration and an output produces the new configuration. The Turing machine itself works as a program which recursively applies transitions and stops when it reaches an accepting state:

```
let rec run m = if accept (cur m) then return m
 else let o = trans (cur m) (tape m (pos m)) in run (update m o)
```

For the deterministic case, we cannot define this function in general in Coq because there is no way to guarantee the termination of the fixpoint. In Alea however, the type `distr` A includes sub-probabilities ($\mu\ P\ 1 < 1$) which models possibly non terminating programs. This type is equipped with a complete partial order structure and thus any monotonic continuous operator of type $(A \to \texttt{distr}\ B) \to A \to \texttt{distr}\ B$ has a fixpoint. In order to model probabilistic Turing machines, we just have to see the transition function as a probabilistic transformation: $\texttt{trans} : \Sigma \to A \to \texttt{distr output}$ and interpret the previous definition of `run` as a function of type $\texttt{config} \to \texttt{distr config}$ using the monadic constructions for `return` and `let` and the general fixpoint on distributions. We can then measure the probability for the machine to terminate from an initial configuration m_0: it is given by the expression $\mu\ (\texttt{run}\, m_0)\ 1$.

In the development of the watermarking algorithms, we only define simple probabilistic programs using structural recursion. However, the robustness of a watermarking scheme will be established using a quantification over all possible attackers. It is important to make sure that our formalisation covers everybody. The main restriction of our model is that random primitives are limited to discrete distributions. However this is sufficient for modelling probabilistic Turing machines, because we only need to express a distribution on outputs which is a finite set. Thus, our formalisation in Coq is adequate: any probabilistic attack can be expressed in our framework, and is thus covered by our results.

2.2 Reasoning with Alea

We shall describe a simple proof, establishing that some function implements the uniform probability law on bit vectors of a given length. We define the `BVrandom` function which given a natural number n, computes uniformly a bit vector of size n. It is defined by structural induction on n, using the `Flip` program which returns `true` with probability $\frac{1}{2}$ and `false` otherwise:

```
Fixpoint BVrandom n : distr (Bvector n) :=
  match n with
  | 0  ⇒ return Bnil
  | Sm ⇒ let hd = Flip in let tl = BVrandom m in return (Vcons hd tl)
  end
```

Then we seek to show that `BVrandom` implements a uniform distribution which means that the probability that `BVrandom` n returns a particular vector x of length n is 2^{-n}. Formally, it is written as follows where `BVeq` is a boolean function testing the equality of two bit vectors and `B2U` is the conversion of booleans to 0 and 1 in the set `U`.

```
Theorem BVrandom_eq : ∀n (x : Bvector n),
   μ (BVrandom n) (fun y ⇒ B2U (BVeq y x)) == (1/2)^n.
```

The interpretation of randomised programs as measures gives us a direct way to establish such a result by using simple algebraic transformations following the structure of the program. Examples of transformations are:

- μ (**return** x) $f == f\ x$,
- μ `Flip` $f == (1/2) \times f$ `true` $+ (1/2) \times f$ `false`,
- μ (**let** $x = P$ **in** $Q\ x$) $f == \mu\ P$ (**fun** $x \Rightarrow \mu\ (Q\ x)\ f$)

The `BVrandom` function is defined by structural induction on n, thus we naturally reason on it by induction on n or (preferably) on the vector x. The base case is trivial: `BVrandom` 0 computes to **return** `Bnil` and both sides simplify to 1, so we can invoke `reflexivity`. When the list has a head and tail, we unfold the computation of `BVrandom (S` n`)` and simplify the obtained measure to make μ (`BVrandom` n) f (almost) appear as:

$$(1/2) \times \mu\ (\texttt{BVrandom}\ n)\ (\textbf{fun}\ y \Rightarrow \texttt{B2U}\ (\texttt{BVeq}\ (\texttt{Vcons true}\ y)\ (\texttt{Vcons}\ a\ x)))+$$
$$(1/2) \times \mu\ (\texttt{BVrandom}\ n)\ (\textbf{fun}\ y \Rightarrow \texttt{B2U}\ (\texttt{BVeq}\ (\texttt{Vcons false}\ y)(\texttt{Vcons}\ a\ x)))$$
$$== (1/2)^{\texttt{S}\ n}$$

We can conclude by doing a case analysis to compare the heads of those lists, followed by the invocation of the induction hypothesis, some simplifications (notably killing the branch where a is not the correct head) and finally we obtain equalities on measures and on reals in `U` which are solved automatically.

In the rest of the paper, we omit most occurrences of `B2U`, implicitly treat decidable propositions (*e*.g., equality on bit vectors) as booleans, and we simply write $\mathcal{P}(e)$ for $\mu\ e$ `B2U`, which represents the probability that e returns `true` when e is a probabilistic program computing a boolean, *i*.e., a Coq object of type `distr bool`.

3 Proof of Robustness

We shall now present our results. They have been fully formalised using Coq version 8.3, the AAC plugin [6] for reasoning modulo associativity commutativity, and version 7 of the ALEA library which benefited from several contributions as part of this work. The full development can be downloaded or replayed online at `http://www.lix.polytechnique.fr/~dbaelde/watermarking/`.

Our proof follows the main language-based technique for proving properties against arbitrary attackers: we reduce robustness to the simpler property of secrecy of the key. In the following, we detail the main steps of our proof for the bit flipping scheme: we first quantify secrecy, then use an efficient reduction of robustness to secrecy, and finally show that the obtained bound is exponentially small in the size of the support. For each step, we provide the informal argument and discuss how this argument carries to the Coq formalisation. Finally, we observe that the substitution algorithm can be treated in the exact same fashion.

3.1 Operations on Bit Vector

Since our algorithms rely heavily on bit vectors, we had to develop libraries for common bit vector operations, both deterministic and probabilistic. This part of our formalisation is mostly straightforward. It benefited a lot from reasoning modulo associativity and commutativity, and we also used type classes to reflect the boolean algebra structure that bit vectors inherit from booleans.

In this paper, we shall use the following notations, which depart slightly from our Coq notations. If X and Y are bit vectors of same length, $X \oplus Y$, $X \mathbin{\&} Y$ and $X \mathbin{\|} Y$ respectively denote bit-wise xor, conjunction and disjunction, and $\neg X$ is the bit-wise negation of X. The number of bits set to 1 in X is $|X|_1$. The Hamming distance `dist` X Y is defined as $|X \oplus Y|_1$. We also define $|X \setminus Y|_1$ as $|X \mathbin{\&} \neg Y|_1$: it counts the number of bits set in X but not in Y.

We defined a couple of uniform random generators: `BVrandom` n returns a bit vector of length n; `BVrandom_k` k n returns a vector of length n with k bits set; `BVrandom_k_mask` k M returns a vector of the same length as the mask M, having k bits set, only at positions where M is also set. This last function is used to randomly modify a vector, in a uniform way that is interesting to consider more closely.

Lemma 1 (`RandomBV.correct_eq`). *We define a correction function that takes a vector X, removes i bits among those set in X and adds m among those not set.*

$$\texttt{correct}\ X\ i\ m := \begin{cases} \textbf{let}\ I = \texttt{BVrandom\_k\_mask}\ i\ X\ \textbf{in} \\ \textbf{let}\ M = \texttt{BVrandom\_k\_mask}\ m\ (\neg X)\ \textbf{in} \\ \quad \textbf{return}\ ((X \mathbin{\&} \neg I) \mathbin{\|} M) \end{cases}$$

For any bit vectors K and K' of length n, `correct` K' $|K' \setminus K|_1$ $|K \setminus K'|_1$ *returns K with the following probability:*

$$\binom{|K'|_1}{|K' \setminus K|_1}^{-1} \binom{n - |K'|_1}{|K \setminus K'|_1}^{-1}$$

Proof. Our lemma simply says that choices of I and M in the correction are independent, and thus the probability of success is the product of the probabilities of finding the two right vectors. However, the corresponding Coq proof turns out to be quite interesting. The key point is that we do not use conditional probabilities to express independence, but rely instead on the structure of the distribution (that is, the probabilistic program) to make independence explicit.

After unfolding a few constructs, and writing i for $|K' \setminus K|_1$ and m for $|K \setminus K'|_1$, the probability that we are considering is the following:

$$\begin{aligned} &\mu\ (\texttt{BVrandom\_k\_mask}\ i\ K')\ (\lambda I. \\ &\quad \mu\ (\texttt{BVrandom\_k\_mask}\ m\ (\neg K'))\ (\lambda M. \\ &\quad\quad K = (K' \mathbin{\&} \neg I) \mathbin{\|} M)) \end{aligned}$$

One can show that `BVrandom_k_mask` k X always ranges among vectors Y such that $|Y \setminus X|_1 = 0$. This can be used (twice) to enrich our expression:

$$\begin{aligned} &\mu\ (\texttt{BVrandom\_k\_mask}\ i\ K')\ (\lambda I.\ (|I \setminus K'|_1 = 0) \times \\ &\quad \mu\ (\texttt{BVrandom\_k\_mask}\ m\ (\neg K'))\ (\lambda M.\ (|M \setminus (\neg K')|_1 = 0) \times \\ &\quad\quad (K = (K' \mathbin{\&} \neg I) \mathbin{\|} M))) \end{aligned}$$

By the stability of distributions under scalar multiplication, we can push multiplications under μ constructs, at which point we observe that $(|I \setminus K'|_1 = 0) \wedge (|M \setminus (\neg K')|_1 = 0) \wedge (K = (K' \mathbin{\&} \neg I) \parallel M)$ is equivalent to $(I = (K' \mathbin{\&} \neg K)) \wedge (M = (K \mathbin{\&} \neg K'))$. This relies on basic boolean algebra, proved notably by observing that for any two vectors u and v of size n, $u \mathbin{\&} v = 0$ implies $u \mathbin{\&} \neg v = u$. Thus, we have simplified our probability distribution as follows:

$$\begin{array}{l}\mu\ (\texttt{BVrandom\_k\_mask}\ i\ K')\ (\lambda I.\\ \quad \mu\ (\texttt{BVrandom\_k\_mask}\ m\ (\neg K'))\ (\lambda M.\\ \quad\quad (I = (K' \mathbin{\&} \neg K)) \times (M = (K \mathbin{\&} \neg K'))))\end{array}$$

In other words, we have succeeded in splitting the result of our computation in two independent parts, one for each probabilistic variable. Using the stability under multiplication in the opposite direction as before, we can thus split the whole computation in two:

$$\begin{array}{l}\quad \big(\mu\ (\texttt{BVrandom\_k\_mask}\ i\ K')\ (\lambda I.\ (I = (K' \mathbin{\&} \neg K))\big)\\ \times\ \big(\mu\ (\texttt{BVrandom\_k\_mask}\ m\ (\neg K'))\ (\lambda M.\ (M = (K \mathbin{\&} \neg K'))\big)\end{array}$$

From there, it is easy to conclude by uniformity of our choice primitive.

3.2 Bit Flipping Watermarking

We now define the bit flipping watermarking scheme and its two security properties. Our definitions rely on a few parameters, which are considered public: n is the size of the mask and support; k is the number of marked bits; δ is the detection threshold, *i.e.*, a message is considered marked if it has more than δ marked bits; γ is the deformation threshold, *i.e.*, two messages are considered indistinguishable if their Hamming distance is less than γ. Obviously, those four parameters should be strictly positive integers. More assumptions will be introduced after the definitions.

For bit flipping on supports of size n, the key is a mask with only k bits set to 1, marking is done by flipping the bits set in the mask, and the number of marked bits in O' is the number of positions set in K where O' and O differ.

$$\begin{aligned}\texttt{genkey} &:= \texttt{BVrandom\_k}\ n\ k\\ \texttt{mark}\ K\ O &:= K \oplus O\\ \#\texttt{marks}\ O\ K\ O' &:= |K \mathbin{\&} (O \oplus O')|_1\end{aligned}$$

We then give games defining robustness and secrecy. Robustness is the ability to resist unmarking challenges, where the attacker is given a marked message and has to return an unmarked message that is indistinguishable from the marked one:

$$\texttt{unmark}\ \mathcal{A} := \begin{cases}\textbf{let}\ K = \texttt{genkey}\ \textbf{in}\\ \textbf{let}\ O = \texttt{BVrandom}\ n\ \textbf{in}\\ \textbf{let}\ O' = \texttt{mark}\ K\ O\ \textbf{in}\\ \textbf{let}\ O'' = \mathcal{A}\ O'\ \textbf{in}\\ \quad \textbf{return}\ (\texttt{dist}\ O'\ O'' < \gamma\ \&\ \#\texttt{marks}\ O\ K\ O'' < \delta)\end{cases}$$

Secrecy is the impossibility for an attacker to guess the key, given only a marked message:

$$\texttt{find\_mask}\ \mathcal{A} := \begin{cases} \textbf{let } K = \texttt{genkey} \textbf{ in} \\ \textbf{let } O = \texttt{BVrandom}\ n \textbf{ in} \\ \textbf{let } O' = \texttt{mark}\ K\ O \textbf{ in} \\ \textbf{let } K' = \mathcal{A}\ O' \textbf{ in} \\ \quad \textbf{return } (K = K') \end{cases}$$

Finally, we make the following natural assumptions on our parameters:

$$\delta \leq k \quad \wedge \quad \gamma \leq n \quad \wedge \quad k \leq \gamma + \delta \quad \wedge \quad 2k \leq n \tag{1}$$

The first two require that the detection threshold is less than the number of marked bits, and that the deformation threshold is less than the support size. Values of δ and γ outside these ranges would be meaningless; another way to put it is that bounding them by k and n respectively does not change the problem at all. The third equation is more interesting: it states that an attack is possible. Indeed a successful attack must alter more than $k - \delta$ bits. Finally, the last inequality states that less than half of the bits should be marked. This is obviously desirable for a watermarking application. In any case, if more than half of the bits are marked, the problem is more simply attacked by trying to find the unmarked bits and we are back to a situation where the constraint is satisfied.

3.3 Secrecy

We prove that the key does not leak through marking: the knowledge of marked messages does not allow a better attack than the blind attack consisting of guessing the key from scratch. While the proof of this first lemma would be deemed straightforward on paper, its formalisation reveals a few interesting steps.

Lemma 2 (`Xor_analysis.secrecy`)**.** *Guessing the key given a marked support is as hard as a blind guess:*

$$\forall \mathcal{A},\ \mathcal{P}(\texttt{find\_mask}\ \mathcal{A}) \leq \binom{n}{k}^{-1}$$

Proof. By definition, `find_mask` $\mathcal{A}$ is:

$$\mu\ \texttt{genkey}\ (\lambda K.\ \mu\ (\texttt{BVrandom}\ n)\ (\lambda O.\ \mu\ (K \oplus O)\ (\lambda O'.\\ \mu\ (\mathcal{A}\ O')\ (\lambda K'.\ \textbf{return}\ (K = K')))))$$

The key observation is that for any vector K, the distribution $K \oplus$ `BVrandom` n is equal to `BVrandom` n. This is proved in `RandomBV.BVxor_noise` by induction on the size of vectors and straightforward computation on the distributions. Using it, we can rewrite `find_mask` $\mathcal{A}$ as follows:

$$\mu\ \texttt{genkey}\ (\lambda K.\ \mu\ (\texttt{BVrandom}\ n)\ (\lambda O'.\\ \mu\ (\mathcal{A}\ O')\ (\lambda K'.\ \textbf{return}\ (K = K'))))$$

From there, the proof is routine: we simply need to permute the various choices to make it explicit that $\mathcal{A}$ is blindly guessing the outcome of the uniform choice genkey. In ALEA, the permutation of probabilistic let-definitions corresponds to commuting integrals, which is not a trivially valid operation in general. However, it has been proved (in ALEA) to always be admissible for discrete distributions, and our distributions are indeed discrete because their domain itself is so. So we permute the introduction of K after that of O' and K', and use the uniformity of genkey, that is BVrandom_k:

$$\begin{pmatrix} \mu\ (\texttt{BVrandom}\ n)\ (\lambda O'. \\ \mu\ (\mathcal{A}\ O')\ (\lambda K'. \\ \quad \mu\ \texttt{genkey}\ (\lambda K.\ K = K'))) \end{pmatrix} \leq \begin{pmatrix} \mu\ (\texttt{BVrandom}\ n)\ (\lambda O'. \\ \mu\ (\mathcal{A}\ O')\ (\lambda K'. \\ \quad 1/\binom{n}{k})) \end{pmatrix}$$

From there, we conclude easily by simplifying the right hand-side program into $1/\binom{n}{k}$ multiplied by the probabilities that BVrandom n and $\mathcal{A}\ O'$ terminate.

3.4 Robustness

We now proceed to reduce robustness to secrecy. Given an attacker that can unmark messages, we build an attack on the key that runs the attack on the mark, considers the changes as an estimation of the key, and randomly attempts to correct it in order to obtain the secret key. In order to do this efficiently, we first bound the error on the estimated key, and then show that it suffices to guess the error to know how many bits are incorrectly set and how many bits are missing in the estimated key.

For the next two lemmas, we consider a run of an attack on the mark. Let K be a mask, *i*.e., a vector of size n satisfying $|K|_1 = k$. Let O be an arbitrary vector of size n, and O' be mark $K\ O$. Let O'' be another vector of size n, representing the attackers' attempt at removing the mark, and let $k_a =$ dist $O'\ O''$ be the number of attacked bits. We define K' to be $O' \oplus O''$. From the viewpoint of considering this set of changes as an approximation of K, we define the error e to be dist $K\ K'$, the number of incorrect bits k_i to be $|K' \setminus K|_1$ (these are the bits set to 1 in K' but not in K, they have no influence on mark detection but affect distortion) and the number of missing bits k_m to be $|K \setminus K'|_1$ (these are the bits which are 0 in K' but 1 in K so where the mark is not removed). We finally define the number of correct bits k_c to be $k - k_m$ (these are the bits set to 1 in both K and K').

Lemma 3 (Maximum error e_{max}). *If the attack succeeds,* i.*e.,* dist $O'\ O'' < \gamma$ *and* #marks $O\ K\ O'' < \delta$*, then* dist $K\ K' < e_{max}$ *where* $e_{max} := \gamma + 2\delta - k$.

Proof. The hypothesis #marks $O\ K\ O'' < \delta$ gives us $k_m < \delta$. We also observe that dist $K\ K' = k_i + k_m$, and $k_i + k_c = |K'|_1 =$ dist $O'\ O'' < \gamma$. We conclude that dist $K\ K' = (k_i + k_c) - k_c + k_m = (k_i + k_c) + 2k_m - k < \gamma + 2\delta - k$.

In our COQ development, this reasoning is done directly in the proof that the reduction is correct, since it is quite simple, relying only on simple properties of boolean operations and linear arithmetic facts, which are proved automatically using the omega tactic of COQ.

For example, with $\gamma = k$ and $\delta = k/2$, we have $e_{\text{max}} = k$. This is a tight bound, in the sense that it can be reached, but it does not bring much information about K: in general, guessing K from K' with $\texttt{dist}\ K\ K' < k$ is essentially as hard as guessing from scratch the positions of the k bits set in K. Fortunately, we can extract much more information, as the next lemma shows.

Lemma 4 (`Xor_analysis.ki_km`). *The quantities k_i and k_m can be derived from k_a and e, since we have $k_i = \frac{e+k_a-k}{2}$ and $k_m = \frac{e+k-k_a}{2}$.*

Proof. Follows immediately from $k_a = k_i + k_c = k_i + (k - k_m)$ and $e = k_i + k_m$.

Definition 1 (Reduction). *Given an attack $\mathcal{A}$ on the mark, we build an attack on the key by starting with the estimated key K' corresponding to the attack on the mark, guessing randomly an error $0 \le e < e_{max}$, deducing k_i and k_m, and guessing the correction of K' according to those parameters.*

$$\mathcal{A}'\ \mathcal{A}\ O' \overset{def}{=} \begin{cases} \textbf{let } O'' = \mathcal{A}\ O' \textbf{ in} \\ \textbf{let } K' = O' \oplus O'' \textbf{ in} \\ \textbf{let } k_a = \texttt{dist}\ O'\ O'' \textbf{ in} \\ \textbf{let } e = \texttt{random\_int}\ e_{max} \textbf{ in} \\ \textbf{let } k_i = (e + k_a - k)/2 \textbf{ in} \\ \textbf{let } k_m = (e + k - k_a)/2 \textbf{ in} \\ \quad \textbf{return } (\texttt{correct}\ K'\ k_i\ k_m) \end{cases}$$

Note that $\mathcal{A}'$ can be enhanced to not consider values of e which yield non-integer k_i and k_m. We do not care to do it here since this would not change asymptotic results.

Lemma 5 (`Xor_analysis.reduction`)

$$\forall \mathcal{A}.\ \mathcal{P}(\texttt{unmark}\ \mathcal{A}) \le e_{max} \times \binom{\gamma}{\lceil \gamma/2 \rceil} \times \binom{n-k+\delta}{\delta} \times \mathcal{P}(\texttt{find\_mask}\ (\mathcal{A}'\ \mathcal{A}))$$

Proof. We consider an execution, introducing O, K, O', then executing $\mathcal{A}'\ \mathcal{A}\ O'$. With probability $\mathcal{P}(\texttt{unmark}\ \mathcal{A})$ we'll obtain an unmarked copy O'' from $\mathcal{A}\ O'$. Then, the correct error $\texttt{dist}\ K\ K'$ will be guessed with a probability of one in e_{max}, and the correct values for k_i and k_c will follow from it. Finally, $\texttt{correct}\ K'\ k_i\ k_m$ will return K with a probability of one in

$$\binom{k_a}{k_i} \times \binom{n-k_a}{k_m}$$

It only remains to bound those two terms independently of k_i, k_m and k_a, using monotonicity properties of the combinatorial function and assumptions on our parameters and on the success of the unmarking attack.

Theorem 1 (`Xor_analysis.robustness`)

$$\forall \mathcal{A}.\ \mathcal{P}(\texttt{unmark}\ A) \le \frac{e_{max} \times \binom{\gamma}{\lfloor \gamma/2 \rfloor} \times \binom{n-k+\delta}{\delta}}{\binom{n}{k}}$$

Proof. Immediate from Lemmas 2 and 5.

3.5 Asymptotic Behaviour

We now show that the bound derived in Theorem 1 is negligible, *i.e.*, that it is eventually bounded by a negative exponential. This standard practice with security bounds is less relevant in watermarking than, for example, in cryptography, because the size of our secret key is fixed. Thus, a precise expression of the bound may be more useful in practice than an asymptotic over-approximation. Nevertheless, addressing the latter is an interesting challenge in terms of formal proofs. In order to achieve this, we move from $[0;1]$ to positive real numbers, which are better behaved and notably form a semi-ring. Specifically, we use a new experimental formalisation of $\mathbb{R}^+$ built on top of `U` in which a real number is represented as a pair with an integral part in `nat` and a fractional part in `U`.

In the following, we are going to study the behaviour of our bound for n large enough. To do so, we need some information on how other parameters grow with n. We shall essentially assume that those parameters are linear in n, which is the standard choice in practice. In addition to (1), we make the following assumptions for some $0 < \alpha < 1$, $c \in \mathbb{N}$ and $e, g \in \mathbb{R}^+$ with non-null c and g:

$$k/n \le \alpha \qquad (2) \qquad\qquad e\mathrm{max} \le e \times n \qquad (4)$$

$$(n/k)^c \ge 4 \qquad (3) \qquad\qquad k - \delta \ge n \times g + \lfloor \gamma/2 \rfloor \times c \qquad (5)$$

The last assumption is formulated in an ad-hoc fashion, but it essentially requires that the difficulty of the attack is large enough and growing linearly with n. Indeed, it quantifies the gap between the maximum number of bits that can be attacked (γ) and the minimum that has to be changed in order to evade detection ($k - \delta$).

When all parameters are linear, we can always determine α, c and e. However, a suitable value for g in (5) can only be found when $k - \delta > \lfloor \gamma/2 \rfloor \times c$. For example, suppose we want to mark $k = \lfloor n/100 \rfloor$ bits with a detection threshold $\delta = \lfloor n/200 \rfloor$. We can take $c = \log(4)/\log(100)$, and assumption (5) roughly requires that γ is less than six times $k - \delta$. Thus, our hypotheses still allow us to cover a satisfyingly wide range of attacks. Our asymptotic bound will become tighter when the attack is made more difficult, *i.e.*, γ is made smaller relative to $k - \delta$.

Lemma 6 (`Asymptotic.final`)**.** *There exists $\beta \in \mathbb{R}^{+*}$ and $m \in \mathbb{N}$ such that for any $n \ge m$,*

$$\frac{emax \times \binom{\gamma}{\lfloor \gamma/2 \rfloor} \times \binom{n-k+\delta}{\delta}}{\binom{n}{k}} \le \alpha^{\lfloor \beta n \rfloor}$$

Proof. We first show that:

$$e\mathrm{max} \times \binom{\gamma}{\lfloor \gamma/2 \rfloor} \times \frac{\binom{n-k+\delta}{\delta}}{\binom{n}{k}} \le e\mathrm{max} \times (2 \times 4^{\lfloor \gamma/2 \rfloor}) \times (\frac{k}{n})^{k-\delta}$$

This is proved by approximating separately each term. For the second term, we use the following precise bound: $\forall k, \binom{2k}{k} \leq 4^k$. For the third term, we show that

$$\frac{\binom{n-k+\delta}{\delta}}{\binom{n}{k}} = \frac{\delta+1 \times \ldots \times k}{(n-k+\delta+1) \times \ldots \times n} = \frac{\delta+1}{n-k+\delta+1} \times \frac{\delta+2}{n-k+\delta+2} \times \ldots \times \frac{k}{n}$$

and we conclude by observing that we have $k-\delta$ increasing factors.

We choose β to be $g/2$, and m is chosen to be large enough so that $2 \times e_{\max} \leq \alpha^{-\lfloor gn/2 \rfloor}$, which can always be obtained since $e_{\max}$ is only linear in n by (4). We obtain the following bound: $\alpha^{-\lfloor gn/2 \rfloor} \times 4^{\lfloor \gamma/2 \rfloor} \times (\frac{k}{n})^{k-\delta}$. By assumption (3) we can further enlarge this into $\alpha^{-\lfloor gn/2 \rfloor} \times (k/n)^{-c\lfloor \gamma/2 \rfloor} \times (k/n)^{k-\delta}$. Using (5) we obtain $\alpha^{-\lfloor gn/2 \rfloor} \times (k/n)^{\lfloor ng \rfloor}$ and we finally obtain $\alpha^{\lfloor gn/2 \rfloor}$ by (2).

Corollary 1 (`Asymptotic.robustness`). *Provided that assumptions (1-5) hold, there exists $\beta > 0$ and $m \in \mathbb{N}$ such that for any $n \geq m$,*

$$\forall \mathcal{A}.\ \mathcal{P}(\texttt{unmark}\ A) \leq \alpha^{\lfloor \beta n \rfloor}$$

3.6 Substitution Watermarking

We now consider marking by substitution. In this scheme, the secret key is made of two parts: a mask M and a message K, both of the same length as the support to be marked. Instead of flipping bits according to the mask as in the previous scheme, bits of the support are replaced by those of the message at positions indicated by the mask. Although the message contains irrelevant information (at positions not set in the mask) in this presentation, this is not a problem for our development.

$$\texttt{genkey} := \begin{cases} \textbf{let } M = \texttt{BVrandom\_k}\ n\ k \textbf{ in} \\ \textbf{let } K = \texttt{BVrandom}\ n \textbf{ in} \\ \quad \textbf{return } (M, K) \end{cases}$$

$$\texttt{mark}\ (M,K)\ O := (O \,\&\, \neg M) \,\|\, (K \,\&\, M)$$

$$\texttt{\#marks}\ (M,K)\ O' := |M \,\&\, \neg(K \oplus O')|_1$$

Note that this scheme is *blind*, *i.e.*, the detection method does not need to refer to the initial support O as before. This property is the reason why substitution is preferred to bit flipping in practice. Indeed, blind schemes allow to delegate the search of marked copies to several agents without having to communicate much information to those agents: When the secret key is generated using a pseudo-random number generator (PRNG) it suffices to communicate the seed that was used to initialise it. Of course, this compression advantage disappears in our theoretical setting; we discuss this in conclusion.

The definition of the robustness game `unmark` is the same as before, and it turns out that we can derive the exact same bound as for bit-flipping watermarking, under the same hypotheses on parameters n, k, δ and γ. The reason is that although there is more information in the secret key, it suffices to guess the mask M to obtain the (useful part of) the message K. Therefore the problem is exactly the same as for bit flipping, the same reduction applies and we obtain the same bound. Formally, the only change is in the definition of the secrecy game:

$$\texttt{find\_mask}\ \mathcal{A} := \begin{cases} \textbf{let}\ (M,K) = \texttt{genkey}\ \textbf{in} \\ \textbf{let}\ O = \texttt{BVrandom}\ n\ \textbf{in} \\ \textbf{let}\ O' = \texttt{mark}\ (M,K)\ O\ \textbf{in} \\ \textbf{let}\ M' = \mathcal{A}\ O'\ \textbf{in} \\ \quad \textbf{return}\ (M = M') \end{cases}$$

Lemma 7 **(**`Substitution_analysis.secrecy`**)**

$$\forall \mathcal{A},\ \mathcal{P}(\texttt{find\_mask}\ \mathcal{A}) \leq \binom{n}{k}^{-1}$$

Lemma 8 **(**`Substitution_analysis.reduction`**).** *Using $\mathcal{A}'$ in Definition [1]:*

$$\forall \mathcal{A}.\ \mathcal{P}(\texttt{unmark}\ A) \leq e_{max} \times \binom{\gamma}{\lceil \gamma/2 \rceil} \times \binom{n-k+\delta}{\delta} \times \mathcal{P}(\texttt{find\_mask}\ (\mathcal{A}'\ \mathcal{A}))$$

4 Conclusion

We have given a robustness result for bit-flipping and substitution schemes against arbitrary attackers, based on a new reduction to secrecy. This reduction and the associated proofs have been fully formalised in Coq on top of the Alea library. Our work is one of the few large examples of using this library, illustrating most of its key aspects with the notable exception of non-terminating probabilistic functions and fixpoints of (sub)distributions. In itself, our development totals around 900 lines of specification and 1500 lines of proofs. But it also required further developments of Alea, including properties of binomial coefficients, a few dedicated tactics to simplify expressions involving distributions, the integration of material about discrete distributions adapted from Certicrypt and the development of the library on positive real numbers (1100 lines of spec and 1700 lines of proofs). Ignoring all the infrastructure work, the core of our development is satisfyingly concise: each of the two robustness proof has only about 200 lines of proofs that follow quite closely the informal presentation.

Our work could be pushed further in several directions. Now that basic watermarking primitives have been formalised and proved robust, one could consider formalising and proving more complex schemes and protocols. For instance, one could prove the security of complete watermarking protocols built from robust NREs. In order to formalise existing proofs in that domain [9], on needs notions such as oracles and polynomial time attackers, and therefore we would naturally turn to the more complex framework of Certicrypt. But there are also fundamental questions which remain open even at the level of the basic embedding primitives. In some contexts, it is relevant to consider repeated attacks, and one should study the advantage that attackers can gain by having access to multiple supports marked with the same key, or multiple marked copies of the same support with different keys. A related question will be to study the impact on robustness of non-uniform distributions for supports, messages, and of other distortion measures than Hamming distances. For example, in the context of watermarking numerical databases, the meaningful notion of distance is based on query answers, which opens up the possibility of *subsetting attacks* that consist in removing a few lines or columns from the database. There are techniques to prevent such attacks, that could be formalised in our framework. Finally, a very important question is the issue of pseudo-random number generators. To the best of our knowledge, there is no formal security work that takes pseudo-random generators into account. This is not a problem in many applications, since true randomness becomes increasingly accessible from physical devices. However, as discussed in the substitution watermarking section, pseudo-random generators are used as a compression device in watermarking. Evaluating the security impact of this practice is thus unavoidable, and promises to be very difficult.

References

1. Adelsbach, A., Katzenbeisser, S., Sadeghi, A.-R.: A Computational Model for Watermark Robustness. In: Camenisch, J.L., Collberg, C.S., Johnson, N.F., Sallee, P. (eds.) IH 2006. LNCS, vol. 4437, pp. 145–160. Springer, Heidelberg (2007)
2. Agrawal, R., Haas, P.J., Kiernan, J.: Watermarking Relational Data: Framework, Algorithms and Analysis. VLDB J. 12(2), 157–169 (2003)
3. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in COQ. MPC 2006 74(8), 568–589 (2009); A preliminary version appeared in the Proc. of MPC 2006
4. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 90–101. ACM (2009)
5. Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: Computer-Aided Security Proofs for the Working Cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
6. Braibant, T., Pous, D.: Tactics for Reasoning Modulo AC in COQ. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 167–182. Springer, Heidelberg (2011)
7. Cox, I.J., Miller, M.L., Bloom, J.A.: Digital Watermarking. Morgan Kaufmann Publishers, Inc., San Francisco (2001)
8. Hasan, O., Tahar, S.: Probabilistic Analysis Using Theorem Proving. VDM Verlag (2008)
9. Hopper, N.J., Molnar, D., Wagner, D.: From Weak to Strong Watermarking. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 362–382. Springer, Heidelberg (2007)
10. Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD thesis, University of Cambridge (2002)
11. Katzenbeisser, S.: A computational model for digital watermarks. In: Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2005) (2005)
12. Khanna, S., Zane, F.: Watermarking maps: hiding information in structured data. In: SODA, pp. 596–605 (2000)

Priority Inheritance Protocol Proved Correct

Xingyuan Zhang[1], Christian Urban[2], and Chunhan Wu[1]

[1] PLA University of Science and Technology, China
[2] King's College London, United Kingdom

Abstract. In real-time systems with threads, resource locking and priority scheduling, one faces the problem of Priority Inversion. This problem can make the behaviour of threads unpredictable and the resulting bugs can be hard to find. The Priority Inheritance Protocol is one solution implemented in many systems for solving this problem, but the correctness of this solution has never been formally verified in a theorem prover. As already pointed out in the literature, the original informal investigation of the Property Inheritance Protocol presents a correctness "proof" for an *incorrect* algorithm. In this paper we fix the problem of this proof by making all notions precise and implementing a variant of a solution proposed earlier. Our formalisation in Isabelle/HOL uncovers facts not mentioned in the literature, but also shows how to efficiently implement this protocol. Earlier correct implementations were criticised as too inefficient. Our formalisation is based on Paulson's inductive approach to verifying protocols.

Keywords: Priority Inheritance Protocol, formal correctness proof, real-time systems, Isabelle/HOL.

1 Introduction

Many real-time systems need to support threads involving priorities and locking of resources. Locking of resources ensures mutual exclusion when accessing shared data or devices that cannot be preempted. Priorities allow scheduling of threads that need to finish their work within deadlines. Unfortunately, both features can interact in subtle ways leading to a problem, called *Priority Inversion*. Suppose three threads having priorities H(igh), M(edium) and L(ow). We would expect that the thread H blocks any other thread with lower priority and the thread itself cannot be blocked indefinitely by threads with lower priority. Alas, in a naive implementation of resource locking and priorities this property can be violated. For this let L be in the possession of a lock for a resource that H also needs. H must therefore wait for L to exit the critical section and release this lock. The problem is that L might in turn be blocked by any thread with priority M, and so H sits there potentially waiting indefinitely. Since H is blocked by threads with lower priorities, the problem is called Priority Inversion. It was first described in [5] in the context of the Mesa programming language designed for concurrent programming.

If the problem of Priority Inversion is ignored, real-time systems can become unpredictable and resulting bugs can be hard to diagnose. The classic example where this happened is the software that controlled the Mars Pathfinder mission in 1997 [9]. Once the spacecraft landed, the software shut down at irregular intervals leading to loss of project time as normal operation of the craft could only resume the next day (the mission and data already collected were fortunately not lost, because of a clever system design). The reason for the shutdowns was that the scheduling software fell victim to Priority Inversion: a low priority thread locking a resource prevented a high priority thread from running in time, leading to a system reset. Once the problem was found, it was rectified by enabling the *Priority Inheritance Protocol* (PIP) [11][1] in the scheduling software.

The idea behind PIP is to let the thread L temporarily inherit the high priority from H until L leaves the critical section unlocking the resource. This solves the problem of H having to wait indefinitely, because L cannot be blocked by threads having priority M. While a few other solutions exist for the Priority Inversion problem, PIP is one that is widely deployed and implemented. This includes VxWorks (a proprietary real-time OS used in the Mars Pathfinder mission, in Boeing's 787 Dreamliner, Honda's ASIMO robot, etc.), but also the POSIX 1003.1c Standard realised for example in libraries for FreeBSD, Solaris and Linux.

One advantage of PIP is that increasing the priority of a thread can be dynamically calculated by the scheduler. This is in contrast to, for example, *Priority Ceiling* [11], another solution to the Priority Inversion problem, which requires static analysis of the program in order to prevent Priority Inversion. However, there has also been strong criticism against PIP. For instance, PIP cannot prevent deadlocks when lock dependencies are circular, and also blocking times can be substantial (more than just the duration of a critical section). Though, most criticism against PIP centres around unreliable implementations and PIP being too complicated and too inefficient. For example, Yodaiken writes in [15]:

> *"Priority inheritance is neither efficient nor reliable. Implementations are either incomplete (and unreliable) or surprisingly complex and intrusive."*

He suggests avoiding PIP altogether by designing the system so that no priority inversion may happen in the first place. However, such ideal designs may not always be achievable in practice.

In our opinion, there is clearly a need for investigating correct algorithms for PIP. A few specifications for PIP exist (in English) and also a few high-level descriptions of implementations (e.g. in the textbook [12, Section 5.6.5]), but they help little with actual implementations. That this is a problem in practice is proved by an email by Baker, who wrote on 13 July 2009 on the Linux Kernel mailing list:

> *"I observed in the kernel code (to my disgust), the Linux PIP implementation is a nightmare: extremely heavy weight, involving maintenance of a full wait-for graph, and requiring updates for a range of events, including priority changes and interruptions of wait operations."*

The criticism by Yodaiken, Baker and others suggests another look at PIP from a more abstract level (but still concrete enough to inform an implementation), and makes PIP a good candidate for a formal verification. An additional reason is that the original presentation of PIP [11], despite being informally "proved" correct, is actually *flawed*.

[1] Sha et al. call it the *Basic Priority Inheritance Protocol* [11] and others sometimes also call it *Priority Boosting* or *Priority Donation*.

Yodaiken [15] points to a subtlety that had been overlooked in the informal proof by Sha et al. They specify in [11] that after the thread (whose priority has been raised) completes its critical section and releases the lock, it "returns to its original priority level." This leads them to believe that an implementation of PIP is "rather straightforward" [11]. Unfortunately, as Yodaiken points out, this behaviour is too simplistic. Consider the case where the low priority thread L locks *two* resources, and two high-priority threads H and H' each wait for one of them. If L releases one resource so that H, say, can proceed, then we still have Priority Inversion with H' (which waits for the other resource). The correct behaviour for L is to switch to the highest remaining priority of the threads that it blocks. The advantage of formalising the correctness of a high-level specification of PIP in a theorem prover is that such issues clearly show up and cannot be overlooked as in informal reasoning (since we have to analyse all possible behaviours of threads, i.e. *traces*, that could possibly happen).

Contributions: There have been earlier formal investigations into PIP [2,4,14], but they employ model checking techniques. This paper presents a formalised and mechanically checked proof for the correctness of PIP (to our knowledge the first one). In contrast to model checking, our formalisation provides insight into why PIP is correct and allows us to prove stronger properties that, as we will show, can help with an efficient implementation of PIP in the educational PINTOS operating system [8]. For example, we found by "playing" with the formalisation that the choice of the next thread to take over a lock when a resource is released is irrelevant for PIP being correct—a fact that has not been mentioned in the literature and not been used in the reference implementation of PIP in PINTOS. This fact, however, is important for an efficient implementation of PIP, because we can give the lock to the thread with the highest priority so that it terminates more quickly.

2 Formal Model of the Priority Inheritance Protocol

The Priority Inheritance Protocol, short PIP, is a scheduling algorithm for a single-processor system.[2] Following good experience in earlier work [13], our model of PIP is based on Paulson's inductive approach to protocol verification [7]. In this approach a *state* of a system is given by a list of events that happened so far (with new events prepended to the list). *Events* of PIP fall into five categories defined as the datatype:

datatype *event*	=	*Create thread priority*	
	|	*Exit thread*	
	|	*Set thread priority*	reset of the priority for *thread*
	|	*P thread cs*	request of resource *cs* by *thread*
	|	*V thread cs*	release of resource *cs* by *thread*

whereby threads, priorities and (critical) resources are represented as natural numbers. The event *Set* models the situation that a thread obtains a new priority given by the programmer or user (for example via the `nice` utility under UNIX). As in

[2] We shall come back later to the case of PIP on multi-processor systems.

Paulson's work, we need to define functions that allow us to make some observations about states. One, called *threads*, calculates the set of "live" threads that we have seen so far:

$$\begin{array}{lcl}
\mathit{threads}\ [] & \overset{\mathit{def}}{=} & \varnothing \\
\mathit{threads}\ (\mathit{Create}\ \mathit{th}\ \mathit{prio}::s) & \overset{\mathit{def}}{=} & \{\mathit{th}\} \cup \mathit{threads}\ s \\
\mathit{threads}\ (\mathit{Exit}\ \mathit{th}::s) & \overset{\mathit{def}}{=} & \mathit{threads}\ s - \{\mathit{th}\} \\
\mathit{threads}\ (\_::s) & \overset{\mathit{def}}{=} & \mathit{threads}\ s
\end{array}$$

In this definition $\_::\_$ stands for list-cons. Another function calculates the priority for a thread *th*, which is defined as

$$\begin{array}{lcl}
\mathit{priority}\ \mathit{th}\ [] & \overset{\mathit{def}}{=} & 0 \\
\mathit{priority}\ \mathit{th}\ (\mathit{Create}\ \mathit{th}'\ \mathit{prio}::s) & \overset{\mathit{def}}{=} & \mathit{if}\ \mathit{th}' = \mathit{th}\ \mathit{then}\ \mathit{prio}\ \mathit{else}\ \mathit{priority}\ \mathit{th}\ s \\
\mathit{priority}\ \mathit{th}\ (\mathit{Set}\ \mathit{th}'\ \mathit{prio}::s) & \overset{\mathit{def}}{=} & \mathit{if}\ \mathit{th}' = \mathit{th}\ \mathit{then}\ \mathit{prio}\ \mathit{else}\ \mathit{priority}\ \mathit{th}\ s \\
\mathit{priority}\ \mathit{th}\ (\_::s) & \overset{\mathit{def}}{=} & \mathit{priority}\ \mathit{th}\ s
\end{array}$$

In this definition we set *0* as the default priority for threads that have not (yet) been created. The last function we need calculates the "time", or index, at which time a process had its priority last set.

$$\begin{array}{lcl}
\mathit{last\_set}\ \mathit{th}\ [] & \overset{\mathit{def}}{=} & 0 \\
\mathit{last\_set}\ \mathit{th}\ (\mathit{Create}\ \mathit{th}'\ \mathit{prio}::s) & \overset{\mathit{def}}{=} & \mathit{if}\ \mathit{th} = \mathit{th}'\ \mathit{then}\ |s|\ \mathit{else}\ \mathit{last\_set}\ \mathit{th}\ s \\
\mathit{last\_set}\ \mathit{th}\ (\mathit{Set}\ \mathit{th}'\ \mathit{prio}::s) & \overset{\mathit{def}}{=} & \mathit{if}\ \mathit{th} = \mathit{th}'\ \mathit{then}\ |s|\ \mathit{else}\ \mathit{last\_set}\ \mathit{th}\ s \\
\mathit{last\_set}\ \mathit{th}\ (\_::s) & \overset{\mathit{def}}{=} & \mathit{last\_set}\ \mathit{th}\ s
\end{array}$$

In this definition $|s|$ stands for the length of the list of events s. Again the default value in this function is *0* for threads that have not been created yet. A *precedence* of a thread *th* in a state s is the pair of natural numbers defined as

$$\mathit{prec}\ \mathit{th}\ s \overset{\mathit{def}}{=} (\mathit{priority}\ \mathit{th}\ s, \mathit{last\_set}\ \mathit{th}\ s)$$

The point of precedences is to schedule threads not according to priorities (because what should we do in case two threads have the same priority), but according to precedences. Precedences allow us to always discriminate between two threads with equal priority by taking into account the time when the priority was last set. We order precedences so that threads with the same priority get a higher precedence if their priority has been set earlier, since for such threads it is more urgent to finish their work. In an implementation this choice would translate to a quite natural FIFO-scheduling of processes with the same priority.

Next, we introduce the concept of *waiting queues*. They are lists of threads associated with every resource. The first thread in this list (i.e. the head, or short *hd*) is chosen to be the one that is in possession of the "lock" of the corresponding resource. We model waiting queues as functions, below abbreviated as *wq*. They take a resource as argument and return a list of threads. This allows us to define when a thread *holds*, respectively *waits* for, a resource *cs* given a waiting queue function *wq*.

$$holds\ wq\ th\ cs \overset{def}{=} th \in set\ (wq\ cs) \wedge th = hd\ (wq\ cs)$$
$$waits\ wq\ th\ cs \overset{def}{=} th \in set\ (wq\ cs) \wedge th \neq hd\ (wq\ cs)$$

In this definition we assume *set* converts a list into a set. At the beginning, that is in the state where no thread is created yet, the waiting queue function will be the function that returns the empty list for every resource.

$$all\_unlocked \overset{def}{=} \lambda\_.\ [] \qquad (1)$$

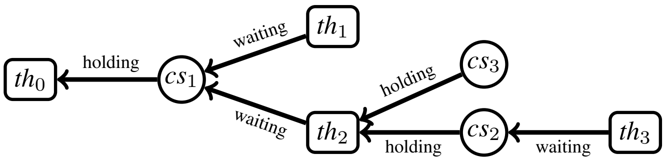
Using *holds* and *waits*, we can introduce *Resource Allocation Graphs* (RAG), which represent the dependencies between threads and resources. We represent RAGs as relations using pairs of the form

$$(T\ th, C\ cs) \quad \text{and} \quad (C\ cs, T\ th)$$

where the first stands for a *waiting edge* and the second for a *holding edge* (C and T are constructors of a datatype for vertices). Given a waiting queue function, a RAG is defined as the union of the sets of waiting and holding edges, namely

$$RAG\ wq \overset{def}{=} \{(T\ th, C\ cs) \mid waits\ wq\ th\ cs\} \cup \{(C\ cs, T\ th) \mid holds\ wq\ th\ cs\}$$

Given four threads and three resources, an instance of a RAG can be pictured as follows:



The use of relations for representing RAGs allows us to conveniently define the notion of the *dependants* of a thread using the transitive closure operation for relations. This gives

$$dependants\ wq\ th \overset{def}{=} \{th' \mid (T\ th', T\ th) \in (RAG\ wq)^+\}$$

This definition needs to account for all threads that wait for a thread to release a resource. This means we need to include threads that transitively wait for a resource being released (in the picture above this means the dependants of th_0 are th_1 and th_2, which wait for resource cs_1, but also th_3, which cannot make any progress unless th_2 makes progress, which in turn needs to wait for th_0 to finish). If there is a circle of dependencies in a RAG, then clearly we have a deadlock. Therefore when a thread requests a resource, we must ensure that the resulting RAG is not circular. In practice, the programmer has to ensure this.

Next we introduce the notion of the *current precedence* of a thread th in a state s. It is defined as

$$cprec\ wq\ s\ th \overset{def}{=} Max\ (\{prec\ th\ s\} \cup \{prec\ th'\ s \mid th' \in dependants\ wq\ th\}) \qquad (2)$$

where the dependants of *th* are given by the waiting queue function. While the precedence *prec* of a thread is determined statically (for example when the thread is created), the point of the current precedence is to let the scheduler increase this precedence, if needed according to PIP. Therefore the current precedence of *th* is given as the maximum of the precedence *th* has in state *s and* all threads that are dependants of *th*. Since the notion *dependants* is defined as the transitive closure of all dependent threads, we deal correctly with the problem in the informal algorithm by Sha et al. [11] where a priority of a thread is lowered prematurely.

The next function, called *schs*, defines the behaviour of the scheduler. It will be defined by recursion on the state (a list of events); this function returns a *schedule state*, which we represent as a record consisting of two functions:

$$(\!| \mathit{wq\_fun}, \mathit{cprec\_fun} |\!)$$

The first function is a waiting queue function (that is, it takes a resource *cs* and returns the corresponding list of threads that lock, respectively wait for, it); the second is a function that takes a thread and returns its current precedence (see the definition in (2)). We assume the usual getter and setter methods for such records.

In the initial state, the scheduler starts with all resources unlocked (the corresponding function is defined in (1)) and the current precedence of every thread is initialised with $(0, 0)$; that means $\mathit{initial\_cprec} \overset{def}{=} \lambda\_.\ (0, 0)$. Therefore we have for the initial shedule state

$$\begin{aligned}&\mathit{schs}\ []\ \overset{def}{=}\\ &\quad (\!| \mathit{wq\_fun} = \mathit{all\_unlocked}, \mathit{cprec\_fun} = \mathit{initial\_cprec} |\!)\end{aligned}$$

The cases for *Create*, *Exit* and *Set* are also straightforward: we calculate the waiting queue function of the (previous) state *s*; this waiting queue function *wq* is unchanged in the next schedule state—because none of these events lock or release any resource; for calculating the next *cprec_fun*, we use *wq* and *cprec*. This gives the following three clauses for *schs*:

$$\begin{aligned}&\mathit{schs}\ (\mathit{Create}\ th\ prio{::}s)\ \overset{def}{=}\\ &\quad \mathit{let}\ wq = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in}\\ &\qquad (\!| \mathit{wq\_fun} = wq, \mathit{cprec\_fun} = \mathit{cprec}\ wq\ (\mathit{Create}\ th\ prio{::}s) |\!)\\ &\mathit{schs}\ (\mathit{Exit}\ th{::}s)\ \overset{def}{=}\\ &\quad \mathit{let}\ wq = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in}\\ &\qquad (\!| \mathit{wq\_fun} = wq, \mathit{cprec\_fun} = \mathit{cprec}\ wq\ (\mathit{Exit}\ th{::}s) |\!)\\ &\mathit{schs}\ (\mathit{Set}\ th\ prio{::}s)\ \overset{def}{=}\\ &\quad \mathit{let}\ wq = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in}\\ &\qquad (\!| \mathit{wq\_fun} = wq, \mathit{cprec\_fun} = \mathit{cprec}\ wq\ (\mathit{Set}\ th\ prio{::}s) |\!)\end{aligned}$$

More interesting are the cases where a resource, say *cs*, is locked or released. In these cases we need to calculate a new waiting queue function. For the event *P th cs*, we have to update the function so that the new thread list for *cs* is the old thread list plus the thread *th* appended to the end of that list (remember the head of this list is assigned to be in the possession of this resource). This gives the clause

$$\begin{array}{l} \mathit{schs}\ (P\ \mathit{th}\ \mathit{cs}{::}s) \overset{\mathit{def}}{=} \\ \quad \mathit{let}\ \mathit{wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in} \\ \quad \mathit{let}\ \mathit{new\_wq} = \mathit{wq}(\mathit{cs} := (\mathit{wq}\ \mathit{cs}\ @\ [\mathit{th}]))\ \mathit{in} \\ \quad\ \ (\!|\mathit{wq\_fun} = \mathit{new\_wq}, \mathit{cprec\_fun} = \mathit{cprec}\ \mathit{new\_wq}\ (P\ \mathit{th}\ \mathit{cs}{::}s)|\!) \end{array}$$

The clause for event *V th cs* is similar, except that we need to update the waiting queue function so that the thread that possessed the lock is deleted from the corresponding thread list. For this list transformation, we use the auxiliary function *release*. A simple version of *release* would just delete this thread and return the remaining threads, namely

$$\begin{array}{lcl} \mathit{release}\ [] & \overset{\mathit{def}}{=} & [] \\ \mathit{release}\ (\_{::}\mathit{qs}) & \overset{\mathit{def}}{=} & \mathit{qs} \end{array}$$

In practice, however, often the thread with the highest precedence in the list will get the lock next. We have implemented this choice, but later found out that the choice of which thread is chosen next is actually irrelevant for the correctness of PIP. Therefore we prove the stronger result where *release* is defined as

$$\begin{array}{lcl} \mathit{release}\ [] & \overset{\mathit{def}}{=} & [] \\ \mathit{release}\ (\_{::}\mathit{qs}) & \overset{\mathit{def}}{=} & \mathit{SOME}\ \mathit{qs}'.\ \mathit{distinct}\ \mathit{qs}' \wedge \mathit{set}\ \mathit{qs}' = \mathit{set}\ \mathit{qs} \end{array}$$

where *SOME* stands for Hilbert's epsilon and implements an arbitrary choice for the next waiting list. It just has to be a list of distinctive threads and contain the same elements as *qs*. This gives for *V* the clause:

$$\begin{array}{l} \mathit{schs}\ (V\ \mathit{th}\ \mathit{cs}{::}s) \overset{\mathit{def}}{=} \\ \quad \mathit{let}\ \mathit{wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in} \\ \quad \mathit{let}\ \mathit{new\_wq} = \mathit{release}\ (\mathit{wq}\ \mathit{cs})\ \mathit{in} \\ \quad\ \ (\!|\mathit{wq\_fun} = \mathit{new\_wq}, \mathit{cprec\_fun} = \mathit{cprec}\ \mathit{new\_wq}\ (V\ \mathit{th}\ \mathit{cs}{::}s)|\!) \end{array}$$

Having the scheduler function *schs* at our disposal, we can "lift", or overload, the notions *waits*, *holds*, *RAG* and *cprec* to operate on states only.

$$\begin{array}{rcl} \mathit{holds}\ s & \overset{\mathit{def}}{=} & \mathit{holds}\ (\mathit{wq\_fun}\ (\mathit{schs}\ s)) \\ \mathit{waits}\ s & \overset{\mathit{def}}{=} & \mathit{waits}\ (\mathit{wq\_fun}\ (\mathit{schs}\ s)) \\ \mathit{RAG}\ s & \overset{\mathit{def}}{=} & \mathit{RAG}\ (\mathit{wq\_fun}\ (\mathit{schs}\ s)) \\ \mathit{cprec}\ s & \overset{\mathit{def}}{=} & \mathit{cprec\_fun}\ (\mathit{schs}\ s) \end{array}$$

With these abbreviations in place we can introduce the notion of a thread being *ready* in a state (i.e. threads that do not wait for any resource) and the running thread.

$$\begin{array}{rcl} \mathit{ready}\ s & \overset{\mathit{def}}{=} & \{\mathit{th} \in \mathit{threads}\ s \mid \forall \mathit{cs}.\ \neg\ \mathit{waits}\ s\ \mathit{th}\ \mathit{cs}\} \\ \mathit{running}\ s & \overset{\mathit{def}}{=} & \{\mathit{th} \in \mathit{ready}\ s \mid \mathit{cprec}\ s\ \mathit{th} = \mathit{Max}\ (\mathit{cprec}\ s\ `\ \mathit{ready}\ s)\} \end{array}$$

In the second definition _ ‘ _ stands for the image of a set under a function. Note that in the initial state, that is where the list of events is empty, the set *threads* is empty and therefore there is neither a thread ready nor running. If there is one or more threads ready, then there can only be *one* thread running, namely the one whose current precedence is equal to the maximum of all ready threads. We use sets to capture both possibilities. We can now also conveniently define the set of resources that are locked by a thread in a given state and also when a thread is detached that state (meaning the thread neither holds nor waits for a resource):

$$\begin{aligned}
&\textit{resources } s\ \textit{th} \overset{\textit{def}}{=} \{cs \mid \textit{holds } s\ \textit{th}\ cs\}\\
&\textit{detached } s\ \textit{th} \overset{\textit{def}}{=} (\nexists cs.\ \textit{holds } s\ \textit{th}\ cs) \wedge (\nexists cs.\ \textit{waits } s\ \textit{th}\ cs)
\end{aligned}$$

The second definition states that *th* in *s*.

Finally we can define what a *valid state* is in our model of PIP. For example we cannot expect to be able to exit a thread, if it was not created yet. These validity constraints on states are characterised by the inductive predicate *step* and *valid_state*. We first give five inference rules for *step* relating a state and an event that can happen next.

$$\frac{\textit{th} \notin \textit{threads } s}{\textit{step } s\ (\textit{Create th prio})} \qquad \frac{\textit{th} \in \textit{running } s \qquad \textit{resources } s\ \textit{th} = \varnothing}{\textit{step } s\ (\textit{Exit th})}$$

The first rule states that a thread can only be created, if it is not alive yet. Similarly, the second rule states that a thread can only be terminated if it was running and does not lock any resources anymore (this simplifies slightly our model; in practice we would expect the operating system releases all locks held by a thread that is about to exit). The event *Set* can happen if the corresponding thread is running.

$$\frac{\textit{th} \in \textit{running } s}{\textit{step } s\ (\textit{Set th prio})}$$

If a thread wants to lock a resource, then the thread needs to be running and also we have to make sure that the resource lock does not lead to a cycle in the RAG. In practice, ensuring the latter is the responsibility of the programmer. In our formal model we brush aside these problematic cases in order to be able to make some meaningful statements about PIP.[3]

$$\frac{\textit{th} \in \textit{running } s \qquad (C\ cs, T\ th) \notin (\textit{RAG } s)^{+}}{\textit{step } s\ (P\ \textit{th}\ cs)}$$

Similarly, if a thread wants to release a lock on a resource, then it must be running and in the possession of that lock. This is formally given by the last inference rule of *step*.

$$\frac{\textit{th} \in \textit{running } s \qquad \textit{holds } s\ \textit{th}\ cs}{\textit{step } s\ (V\ \textit{th}\ cs)}$$

[3] This situation is similar to the infamous *occurs check* in Prolog: In order to say anything meaningful about unification, one needs to perform an occurs check. But in practice the occurs check is omitted and the responsibility for avoiding problems rests with the programmer.

A valid state of PIP can then be conveniently be defined as follows:

$$\frac{}{\textit{valid\_state}\ []} \qquad \frac{\textit{valid\_state}\ s \qquad \textit{step}\ s\ e}{\textit{valid\_state}\ (e{::}s)}$$

This completes our formal model of PIP. In the next section we present properties that show our model of PIP is correct.

3 The Correctness Proof

Sha et al. state their first correctness criterion for PIP in terms of the number of low-priority threads [11, Theorem 3]: if there are n low-priority threads, then a blocked job with high priority can only be blocked a maximum of n times. Their second correctness criterion is given in terms of the number of critical resources [11, Theorem 6]: if there are m critical resources, then a blocked job with high priority can only be blocked a maximum of m times. Both results on their own, strictly speaking, do *not* prevent indefinite, or unbounded, Priority Inversion, because if a low-priority thread does not give up its critical resource (the one the high-priority thread is waiting for), then the high-priority thread can never run. The argument of Sha et al. is that *if* threads release locked resources in a finite amount of time, then indefinite Priority Inversion cannot occur—the high-priority thread is guaranteed to run eventually. The assumption is that programmers must ensure that threads are programmed in this way. However, even taking this assumption into account, the correctness properties of Sha et al. are *not* true for their version of PIP—despite being "proved". As Yodaiken [15] pointed out: If a low-priority thread possesses locks to two resources for which two high-priority threads are waiting for, then lowering the priority prematurely after giving up only one lock, can cause indefinite Priority Inversion for one of the high-priority threads, invalidating their two bounds.

Even when fixed, their proof idea does not seem to go through for us, because of the way we have set up our formal model of PIP. One reason is that we allow critical sections, which start with a P-event and finish with a corresponding V-event, to arbitrarily overlap (something Sha et al. explicitly exclude). Therefore we have designed a different correctness criterion for PIP. The idea behind our criterion is as follows: for all states s, we know the corresponding thread th with the highest precedence; we show that in every future state (denoted by s' @ s) in which th is still alive, either th is running or it is blocked by a thread that was alive in the state s and was waiting for or in the possession of a lock in s. Since in s, as in every state, the set of alive threads is finite, th can only be blocked a finite number of times. This is independent of how many threads of lower priority are created in s'. We will actually prove a stronger statement where we also provide the current precedence of the blocking thread. However, this correctness criterion hinges upon a number of assumptions about the states s and s' @ s, the thread th and the events happening in s'. We list them next:

Assumptions on the states s and s' @ s: We need to require that s and s' @ s are valid states:

valid_state s, valid_state $(s'$ @ $s)$

Assumptions on the thread th: The thread *th* must be alive in *s* and has the highest precedence of all alive threads in *s*. Furthermore the priority of *th* is *prio* (we need this in the next assumptions).

$th \in threads\ s$
$prec\ th\ s = Max\ (cprec\ s\ \text{`}\ threads\ s)$
$prec\ th\ s = (prio, \_)$

Assumptions on the events in s': We want to prove that *th* cannot be blocked indefinitely. Of course this can happen if threads with higher priority than *th* are continuously created in s'. Therefore we have to assume that events in s' can only create (respectively set) threads with equal or lower priority than *prio* of *th*. We also need to assume that the priority of *th* does not get reset and also that *th* does not get "exited" in s'. This can be ensured by assuming the following three implications.

If $Create\ th'\ prio' \in set\ s'$ *then* $prio' \le prio$
If $Set\ th'\ prio' \in set\ s'$ *then* $th' \neq th$ *and* $prio' \le prio$
If $Exit\ th' \in set\ s'$ *then* $th' \neq th$

The locale mechanism of Isabelle helps us to manage conveniently such assumptions [3]. Under these assumptions we shall prove the following correctness property:

Theorem 1. *Given the assumptions about states s and s'* @ *s, the thread th and the events in s', if* $th' \in running\ (s'$ @ $s)$ *and* $th' \neq th$ *then* $th' \in threads\ s$, $\neg\ detached\ s\ th'$ *and* $cprec\ (s'$ @ $s)\ th' = prec\ th\ s$.

This theorem ensures that the thread *th*, which has the highest precedence in the state *s*, can only be blocked in the state s' @ s by a thread th' that already existed in *s* and requested or had a lock on at least one resource—that means the thread was not *detached* in *s*. As we shall see shortly, that means there are only finitely many threads that can block *th* in this way and then they need to run with the same current precedence as *th*.

Like in the argument by Sha et al. our finite bound does not guarantee absence of indefinite Priority Inversion. For this we further have to assume that every thread gives up its resources after a finite amount of time. We found that this assumption is awkward to formalise in our model. Therefore we leave it out and let the programmer assume the responsibility to program threads in such a benign manner (in addition to causing no circularity in the *RAG*). In this detail, we do not make any progress in comparison with the work by Sha et al. However, we are able to combine their two separate bounds into a single theorem improving their bound.

In what follows we will describe properties of PIP that allow us to prove Theorem 1 and, when instructive, briefly describe our argument. It is relatively easy to see that

$running\ s \subseteq ready\ s \subseteq threads\ s$
If *valid_state s* *then* *finite* $(threads\ s)$.

The second property is by induction of *valid_state*. The next three properties are

If valid_state s and waits s th cs_1 *and waits s th* cs_2 *then* $cs_1 = cs_2$.
If holds s th_1 *cs and holds s* th_2 *cs then* $th_1 = th_2$.
If valid_state s and $th_1 \in$ *running s and* $th_2 \in$ *running s then* $th_1 = th_2$.

The first property states that every waiting thread can only wait for a single resource (because it gets suspended after requesting that resource); the second that every resource can only be held by a single thread; the third property establishes that in every given valid state, there is at most one running thread. We can also show the following properties about the *RAG* in *s*.

If valid_state s then:
acyclic $(RAG\ s)$, *finite* $(RAG\ s)$ *and wf* $((RAG\ s)^{-1})$,
if T th $\in$ *Domain* $(RAG\ s)$ *then th* $\in$ *threads s and*
if T th $\in$ *Range* $(RAG\ s)$ *then th* $\in$ *threads s.*

The acyclicity property follows from how we restricted the events in *step*; similarly the finiteness and well-foundedness property. The last two properties establish that every thread in a *RAG* (either holding or waiting for a resource) is a live thread.

The key lemma in our proof of Theorem 1 is as follows:

Lemma 1. *Given the assumptions about states s and* s' @ *s, the thread th and the events in* s', *if* $th' \in$ *threads* $(s'$ @ $s)$, $th' \neq$ *th and detached* $(s'$ @ $s)$ th'
then $th' \notin$ *running* $(s'$ @ $s)$.

The point of this lemma is that a thread different from *th* (which has the highest precedence in *s*) and not holding any resource, cannot be running in the state s' @ *s*.

Proof. Since thread th' does not hold any resource, no thread can depend on it. Therefore its current precedence *cprec* $(s'$ @ $s)$ th' equals its own precedence *prec* th' $(s'$ @ $s)$. Since *th* has the highest precedence in the state $(s'$ @ $s)$ and precedences are distinct among threads, we have *prec* th' $(s'$ @ $s) <$ *prec th* $(s'$ @ $s)$. From this we have *cprec* $(s'$ @ $s)$ $th' <$ *prec th* $(s'$ @ $s)$. Since *prec th* $(s'$ @ $s)$ is already the highest *cprec* $(s'$ @ $s)$ *th* can not be higher than this and can not be lower either (by definition of *cprec*). Consequently, we have *prec th* $(s'$ @ $s) =$ *cprec* $(s'$ @ $s)$ *th*. Finally we have *cprec* $(s'$ @ $s)$ $th' <$ *cprec* $(s'$ @ $s)$ *th*. By defintion of *running*, th' can not be running in state s' @ *s*, as we had to show. □

Since th' is not able to run in state s' @ *s*, it is not able to issue a *P* or *V* event. Therefore if s' @ *s* is extended one step further, th' still cannot hold any resource. The situation will not change in further extensions as long as *th* holds the highest precedence.

From this lemma we can deduce Theorem 1: that *th* can only be blocked by a thread th' that held some resource in state *s* (that is not *detached*). And furthermore that the current precedence of th' in state $(s'$ @ $s)$ must be equal to the precedence of *th* in *s*. We show this theorem by induction on s' using Lemma 1. This theorem gives a stricter bound on the threads that can block *th* than the one obtained by Sha et al. [11]: only threads that were alive in state *s* and moreover held a resource. This means our bound is in terms of both—alive threads in state s and number of critical resources. Finally, the theorem establishes that the blocking threads have the current precedence raised to the precedence of *th*.

We can furthermore prove that under our assumptions no deadlock exists in the state $s' @ s$ by showing that $\mathit{running}\ (s' @ s)$ is not empty.

Lemma 2. *Given the assumptions about states s and* $s' @ s$*, the thread th and the events in* s'*,* $\mathit{running}\ (s' @ s) \neq \emptyset$.

Proof. If *th* is blocked, then by following its dependants graph, we can always reach a ready thread th', and that thread must have inherited the precedence of *th*. □

4 Properties for an Implementation

While our formalised proof gives us confidence about the correctness of our model of PIP, we found that the formalisation can even help us with efficiently implementing it.

For example Baker complained that calculating the current precedence in PIP is quite "heavy weight" in Linux (see the Introduction). In our model of PIP the current precedence of a thread in a state s depends on all its dependants—a "global" transitive notion, which is indeed heavy weight (see Def. shown in (2)). We can however improve upon this. For this let us define the notion of *children* of a thread *th* in a state s as

$$\mathit{children}\ s\ th \overset{def}{=} \{th' \mid \exists cs.\ (T\ th', C\ cs) \in \mathit{RAG}\ s \wedge (C\ cs, T\ th) \in \mathit{RAG}\ s\}$$

where a child is a thread that is only one "hop" away from the thread *th* in the *RAG* (and waiting for *th* to release a resource). We can prove the following lemma.

Lemma 3. *If valid_state s then*

$$\mathit{cprec}\ s\ th = \mathit{Max}\ (\{\mathit{prec}\ th\ s\} \cup \mathit{cprec}\ s\ \text{`}\ \mathit{children}\ s\ th).$$

That means the current precedence of a thread *th* can be computed locally by considering only the children of *th*. In effect, it only needs to be recomputed for *th* when one of its children changes its current precedence. Once the current precedence is computed in this more efficient manner, the selection of the thread with highest precedence from a set of ready threads is a standard scheduling operation implemented in most operating systems.

Of course the main work for implementing PIP involves the scheduler and coding how it should react to events. Below we outline how our formalisation guides this implementation for each kind of events.

Create th prio: We assume that the current state s' and the next state $s \overset{def}{=} \mathit{Create}\ th\ \mathit{prio}{::}s'$ are both valid (meaning the event is allowed to occur). In this situation we can show that

$\mathit{RAG}\ s = \mathit{RAG}\ s'$,
$\mathit{cprec}\ s\ th = \mathit{prec}\ th\ s$, *and*
If $th' \neq th$ *then* $\mathit{cprec}\ s\ th' = \mathit{cprec}\ s'\ th'$.

This means in an implementation we do not have recalculate the *RAG* and also none of the current precedences of the other threads. The current precedence of the created thread *th* is just its precedence, namely the pair $(prio, |s|)$.

Exit th: We again assume that the current state s' and the next state $s \overset{def}{=} \mathit{Exit}\ th{::}s'$ are both valid. We can show that

$RAG\ s = RAG\ s'$, *and*
If $th' \neq th$ *then* $cprec\ s\ th' = cprec\ s'\ th'$.

This means again we do not have to recalculate the *RAG* and also not the current precedences for the other threads. Since *th* is not alive anymore in state *s*, there is no need to calculate its current precedence.

Set th prio: We assume that s' and $s \overset{def}{=} \mathit{Set}\ th\ prio{::}s'$ are both valid. We can show that

$RAG\ s = RAG\ s'$, *and*
If $th' \neq th$ *and* $th \notin dependants\ s\ th'$ *then* $cprec\ s\ th' = cprec\ s'\ th'$.

The first property is again telling us we do not need to change the *RAG*. The second shows that the *cprec*-values of all threads other than *th* are unchanged. The reason is that *th* is running; therefore it is not in the *dependants* relation of any other thread. This in turn means that the change of its priority cannot affect other threads.

V th cs: We assume that s' and $s \overset{def}{=} V\ th\ cs{::}s'$ are both valid. We have to consider two subcases: one where there is a thread to "take over" the released resource *cs*, and one where there is not. Let us consider them in turn. Suppose in state *s*, the thread th' takes over resource *cs* from thread *th*. We can prove

$$RAG\ s = RAG\ s' - \{(C\ cs, T\ th), (T\ th', C\ cs)\} \cup \{(C\ cs, T\ th')\}$$

which shows how the *RAG* needs to be changed. The next lemma suggests how the current precedences need to be recalculated. For threads that are not *th* and th' nothing needs to be changed, since we can show

If $th'' \neq th$ *and* $th'' \neq th'$ *then* $cprec\ s\ th'' = cprec\ s'\ th''$.

For *th* and th' we need to use Lemma 3 to recalculate their current precedence since their children have changed.

In the other case where there is no thread that takes over *cs*, we can show how to recalculate the *RAG* and also show that no current precedence needs to be recalculated.

$RAG\ s = RAG\ s' - \{(C\ cs, T\ th)\}$
$cprec\ s\ th' = cprec\ s'\ th'$

P th cs: We assume that s' and $s \overset{def}{=} P\ th\ cs{::}s'$ are both valid. We again have to analyse two subcases, namely the one where *cs* is not locked, and one where it is. We treat the former case first by showing that

$$\begin{array}{l} \mathit{RAG}\ s = \mathit{RAG}\ s' \cup \{(C\ cs, T\ th)\} \\ \mathit{cprec}\ s\ th' = \mathit{cprec}\ s'\ th' \end{array}$$

This means we need to add a holding edge to the *RAG* and no current precedence needs to be recalculated.

In the second case we know that resource *cs* is locked. We can show that

$$\begin{array}{l} \mathit{RAG}\ s = \mathit{RAG}\ s' \cup \{(T\ th, C\ cs)\} \\ \textit{If } th \notin \mathit{dependants}\ s\ th' \textit{ then } \mathit{cprec}\ s\ th' = \mathit{cprec}\ s'\ th'. \end{array}$$

That means we have to add a waiting edge to the *RAG*. Furthermore the current precedence for all threads that are not dependants of *th* are unchanged. For the others we need to follow the edges in the *RAG* and recompute the *cprec*. To do this we can start from *th* and follow the *RAG*-edges to recompute using Lemma 3 the *cprec* of every thread encountered on the way. Since the *RAG* is loop free, this procedure will always stop. The following lemma shows, however, that this procedure can actually stop often earlier without having to consider all dependants.

$$\begin{array}{l} \textit{If } th \in \mathit{dependants}\ s\ th',\ th' \in \mathit{dependants}\ s\ th'' \textit{ and } \mathit{cprec}\ s\ th' = \mathit{cprec}\ s'\ th' \\ \textit{then } \mathit{cprec}\ s\ th'' = \mathit{cprec}\ s'\ th''. \end{array}$$

This lemma states that if an intermediate *cprec*-value does not change, then the procedure can also stop, because none of its dependent threads will have their current precedence changed.

As can be seen, a pleasing byproduct of our formalisation is that the properties in this section closely inform an implementation of PIP, namely whether the *RAG* needs to be reconfigured or current precedences need to be recalculated for an event. This information is provided by the lemmas we proved. We confirmed that our observations translate into practice by implementing our version of PIP on top of PINTOS, a small operating system written in C and used for teaching at Stanford University [8]. To implement PIP, we only need to modify the kernel functions corresponding to the events in our formal model. The events translate to the following function interface in PINTOS:

Event	**PINTOS function**
Create	*thread_create*
Exit	*thread_exit*
Set	*thread_set_priority*
P	*lock_acquire*
V	*lock_release*

Our implicit assumption that every event is an atomic operation is ensured by the architecture of PINTOS. The case where an unlocked resource is given next to the waiting thread with the highest precedence is realised in our implementation by priority queues. We implemented them as *Braun trees* [6], which provide efficient $O(\log n)$-operations for accessing and updating. Apart from having to implement relatively complex datastructures in C using pointers, our experience with the implementation has been very positive: our specification and formalisation of PIP translates smoothly to an efficent implementation in PINTOS.

5 Conclusion

The Priority Inheritance Protocol (PIP) is a classic textbook algorithm used in many real-time operating systems in order to avoid the problem of Priority Inversion. Although classic and widely used, PIP does have its faults: for example it does not prevent deadlocks in cases where threads have circular lock dependencies.

We had two goals in mind with our formalisation of PIP: One is to make the notions in the correctness proof by Sha et al. [11] precise so that they can be processed by a theorem prover. The reason is that a mechanically checked proof avoids the flaws that crept into their informal reasoning. We achieved this goal: The correctness of PIP now only hinges on the assumptions behind our formal model. The reasoning, which is sometimes quite intricate and tedious, has been checked by Isabelle/HOL. We can also confirm that Paulson's inductive method for protocol verification [7] is quite suitable for our formal model and proof. The traditional application area of this method is security protocols.

The second goal of our formalisation is to provide a specification for actually implementing PIP. Textbooks, for example [12, Section 5.6.5], explain how to use various implementations of PIP and abstractly discuss their properties, but surprisingly lack most details important for a programmer who wants to implement PIP (similarly Sha et al. [11]). That this is an issue in practice is illustrated by the email from Baker we cited in the Introduction. We achieved also this goal: The formalisation allowed us to efficently implement our version of PIP on top of PINTOS [8], a simple instructional operating system for the x86 architecture. It also gives the first author enough data to enable his undergraduate students to implement PIP (as part of their OS course). A byproduct of our formalisation effort is that nearly all design choices for the PIP scheduler are backed up with a proved lemma. We were also able to establish the property that the choice of the next thread which takes over a lock is irrelevant for the correctness of PIP.

PIP is a scheduling algorithm for single-processor systems. We are now living in a multi-processor world. Priority Inversion certainly occurs also there. However, there is very little "foundational" work about PIP-algorithms on multi-processor systems. We are not aware of any correctness proofs, not even informal ones. There is an implementation of a PIP-algorithm for multi-processors as part of the "real-time" effort in Linux, including an informal description of the implemented scheduling algorithm given in [10]. We estimate that the formal verification of this algorithm, involving more fine-grained events, is a magnitude harder than the one we presented here, but still within reach of current theorem proving technology. We leave this for future work.

The most closely related work to ours is the formal verification in PVS of the Priority Ceiling Protocol done by Dutertre [1]—another solution to the Priority Inversion problem, which however needs static analysis of programs in order to avoid it. There have been earlier formal investigations into PIP [2,4,14], but they employ model checking techniques. The results obtained by them apply, however, only to systems with a fixed size, such as a fixed number of events and threads. In contrast, our result applies to systems of arbitrary size. Moreover, our result is a good witness for one of the major reasons to be interested in machine checked reasoning: gaining deeper understanding of the subject matter.

Our formalisation consists of around 210 lemmas and overall 6950 lines of readable Isabelle/Isar code with a few apply-scripts interspersed. The formal model of PIP is 385 lines long; the formal correctness proof 3800 lines. Some auxiliary definitions and proofs span over 770 lines of code. The properties relevant for an implementation require 2000 lines.

Acknowledgements. We are grateful for the comments we received from anonymous referees.

References

1. Dutertre, B.: The Priority Ceiling Protocol: Formalization and Analysis Using PVS. In: Proc. of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS), pp. 151–160. IEEE Computer Society (2000)
2. Faria, J.M.S.: Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC. PhD thesis, University of Porto (2008)
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. Jahier, E., Halbwachs, N., Raymond, P.: Synchronous Modeling and Validation of Priority Inheritance Schedulers. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 140–154. Springer, Heidelberg (2009)
5. Lampson, B.W., Redell, D.D.: Experiences with Processes and Monitors in Mesa. Communications of the ACM 23(2), 105–117 (1980)
6. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press (1996)
7. Paulson, L.C.: The Inductive Approach to Verifying Cryptographic Protocols. Journal of Computer Security 6(1-2), 85–128 (1998)
8. Pfaff, B.: PINTOS, `http://www.stanford.edu/class/cs140/projects/`
9. Reeves, G.E.: Re: What Really Happened on Mars?. Risks Forum 19(54) (1998)
10. Rostedt, S.: RT-Mutex Implementation Design, Linux Kernel Distribution at, `http://www.kernel.org/doc/Documentation/rt-mutex-design.txt`
11. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers 39(9), 1175–1185 (1990)
12. Vahalia, U.: UNIX Internals: The New Frontiers. Prentice-Hall (1996)
13. Wang, J., Yang, H., Zhang, X.: Liveness Reasoning with Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 485–499. Springer, Heidelberg (2009)
14. Wellings, A., Burns, A., Santos, O.M., Brosgol, B.M.: Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java. In: Proc. of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 115–123. IEEE Computer Society (2007)
15. Yodaiken, V.: Against Priority Inheritance. Technical report, Finite State Machine Labs (FSMLabs) (2004)

Formalization of Shannon's Theorems in SSReflect-Coq

Reynald Affeldt and Manabu Hagiwara

Research Institute for Secure Systems,
National Institute of Advanced Industrial Science and Technology, Japan

Abstract. The most fundamental results of information theory are Shannon's theorems. These theorems express the bounds for reliable data compression and transmission over a noisy channel. Their proofs are non-trivial but rarely detailed, even in the introductory literature. This lack of formal foundations makes it all the more unfortunate that crucial results in computer security rely solely on information theory (the so-called "unconditional security"). In this paper, we report on the formalization of a library for information theory in the SSReflect extension of the Coq proof-assistant. In particular, we produce the first formal proofs of the source coding theorem (that introduces the entropy as the bound for lossless compression), and the direct part of the more difficult channel coding theorem (that introduces the capacity as the bound for reliable communication over a noisy channel).

1 Introduction

"Information theory answers two fundamental questions in communication theory: What is the ultimate data compression (answer: the entropy H), and what is the ultimate transmission rate of communication (answer: the channel capacity C)." This is the very first sentence of the reference book on information theory by Cover and Thomas [5]. This paper is precisely about the formalization of Shannon's theorems that answer these two fundamental questions.

The proofs of Shannon's theorems are non-trivial but are rarely detailed (let alone formalized), even in the introductory literature. Shannon's original proofs [1] in 1948 are well-known to be informal; rigorous versions appeared several years later. Even today, the bounds that appear in Shannon's theorems (these theorems are asymptotic) are never made explicit and their existence is seldom proved carefully.

This lack of formal foundations makes it all the more unfortunate that several results in computer security rely crucially on information theory: this is the so-called field of "unconditional security" (one-time pad protocol, evaluation of information leakage, key distribution protocol over a noisy channel, etc.). A formalization of information theory would be a first step towards the verification of cryptographic systems based on unconditional security, and, more generally, for the rigorous design of critical communication devices.

In this paper, our first contribution is to provide a library of formal definitions and lemmas for information theory. First, we formalize finite probability, up to the weak law of large numbers, and apply this formalization to the formalization of basic information-theoretic concepts such as entropy and typical sequences. This line of work has already been investigated by Hasan et al. [6,11,13] and by Coble [9], with the HOL proof-assistants. The originality of our library (besides the fact that we are working with the Coq proof-assistant [10]) lies in the formalization of advanced concepts such as channels, codes, jointly typical sequences, etc., that are necessary to state and prove Shannon's theorems.

Our second and main contribution is to provide the first (to the best of our knowledge) formal proofs of Shannon's theorems. Precisely, we formalize the source coding theorem (direct and converse parts), that introduces the entropy as the bound for lossless compression, and the direct part of the channel coding theorem, that introduces the channel capacity as the bound for reliable communication over a noisy channel.

The formalization of Shannon's theorems is not a trivial matter because, in addition to the complexity of a theorem such as the channel coding theorem, the literature does not provide proofs that are organized in a way that facilitates formalization. Most importantly, it is necessary to rework the proofs so that the (asymptotic) bounds can be formalized. Indeed, information theorists often resort to claims such as "this holds for n sufficiently large", but there are in general several parameters that are working together so that one cannot choose one without checking the others. Another kind of approximation that matters when formalizing is the type of arguments. For example, in the proof of the source coding theorem, it is mathematically important to treat the source rate as a rational and not as a real, but such details are often overlooked. In order to ease formalization, we make several design decisions to reduce the number of concepts involved. For example, we do not use conditional entropy in an explicit way, and, more generally, we avoid explicit use of conditional probabilities, except for the definition of discrete channels. In fact, we believe that this even facilitates informal understanding because proofs are more "to the point".

We carried out formalization in the SSReflect extension [12] of the Coq proof-assistant [10]. This is because information theory involves many calculations with Σ/Π-notations over various kinds of sets (tuples, functions, etc.) for which SSReflect's library (in particular, canonical big operators [7]) are well-suited. Formal definitions and lemmas that appear in this paper are taken directly from the scripts (available at [14]), modulo enhancements with colors and standard non-ASCII characters to improve reading.

Paper Outline. In Sect. 2, we formalize definitions and properties about finite probability to be used in the rest of the paper. In Sect. 3, we introduce the concept of typical sequence. In Sect. 4, we state the source coding theorem and give an outline of the proof of the direct part. In Sect. 5, we formalize the concept of channel and illustrate related definitions thoroughly using the example of the binary symmetric channel. Finally, we state and prove the direct part of the channel coding theorem in Sect. 6. Section 7 is dedicated to related work.

2 The Basics: Finite Probability

We introduce basic definitions about probability (to explain the notations to be used in this paper) and formalize the weak law of large numbers. We do not claim that this formalization is a major contribution in itself because there exist more general formalizations of probability theory (in particular in the HOL proof-assistant [6,11,13]) but providing a new formalization using SSREFLECT will allow us to take advantage of its library to prove Shannon's theorems.

2.1 Probability Distributions

A distribution over a finite type `A` (i.e., of type `finType` in SSREFLECT) is defined as a real-valued probability mass function `pmf` (`R` is the type of reals in Coq standard library) with positive outputs (proof `pmf0` below) that sum to 1 (proof `pmf1`; the big sum operator comes from SSREFLECT [7]):

```
0 Record dist := mkDist {
1   pmf :> A → R ;
2   pmf0 : ∀ a, 0 ≤ pmf a ;
3   pmf1 : Σ_(a ∈ A) pmf a = 1 }.
```

`P : dist A` is a `Record` but, thanks to the coercion line 1, we can write "`P a`" as a function application to represent the probability associated with `a`.

We will be led to define several kinds of distributions in the course of this paper. Here is a first example. Given distributions `P1` over `A` and `P2` over `B`, the *product distribution* `P1` × `P2` over `A * B` is defined as follows:

```
Definition Pprod_dist : dist [finType of A * B].
apply mkDist with (fun x ⇒ P1 x.1 * P2 x.2) ... Defined.
```

(We omit the proofs of `pmf0` and `pmf1` in this paper; the `.1` (resp. `.2`) notation is for the first (resp. second) pair projection); the notation `[finType of ...]` is just a type cast.)

Given a distribution `P` over `A`, the probability of an event (encoded as a boolean predicate of type `pred A`) is defined as follows:

```
Definition Pr (Q : pred A) := Σ_(a ∈ A | Q a) P a.
```

2.2 Random Variables

We formalize a random variable as a distribution coupled with a real-valued function: `Record rvar A := {rv_dist : dist A ; rv_fun :> A →R }.` (This definition is sufficient because `A` is a finite type.) Again, thanks to the coercion, given a random variable `X` and `a` belonging to its sample space, one can write "`X a`" as in standard mathematical writing despite the fact that `X` is actually a `Record`. Furthermore, we note p`_X` the distribution underlying the random variable `X`.

Given a random variable `X` over `A`, and writing `img X` for its image, we define for example the expected value as follows:

```
Definition E := Σ_(r ← img X) r * Pr p_X [pred i | X i =R r].
```

Let us now define the sum of random variables. Below, `n.-tuple A` is the SSReflect type of `n`-tuples over `A` (A^n in standard mathematical writing).

Given distributions `P1` over `A` and `P2` over `n.-tuple A` the *joint distribution* `P` over `n+1.-tuple A` is defined in terms of marginal distributions by the following predicate:

```
Definition joint :=
  (∀ x, P1 x = Σ_(i ∈ {:n+1.-tuple A} | thead i = x) P i) ∧
  (∀ x, P2 x = Σ_(i ∈ {:n+1.-tuple A} | behead i = x) P i).
```

Informally speaking, `joint P1 P2 P` is a relation that defines the distribution `P1` (resp. `P2`) from the distribution `P` by taking into account only the first element (resp. all the elements but the first) of the tuples from the sample space (`thead` returns the first element of a tuple; `behead` returns all the elements but the first).

The random variable `X` is the sum of `X1` and `X2` when the distribution of `X` is the joint distribution of the distributions of `X1` and `X2` and the output of `X` is the sum of the outputs of `X1` and `X2`:

```
Definition sum := joint p_X1 p_X2 p_X ∧
  X =1 [ffun x ⇒ X1 (thead x) + X2 [tuple of (behead x)]].
```

(`[ffun x ⇒...]` is a SSReflect notation to define partial functions over finite domains: `[tuple of ...]` is just a type cast.)

The random variables `X` over `A` and `Y` over `n.-tuple A` are *independent* for a distribution `P` over `n+1.-tuple A` when:

```
Definition inde_rvar := ∀ x y,
  Pr P [pred xy | (X (thead xy) =R x) ∧
                  (Y [tuple of (behead xy)] =R y)] =
  Pr p_X [pred x | X x =R x] * Pr p_Y [pred y | Y x =R y].
```

We define the sum of several random variables by generalizing `sum` to an inductive predicate (like in [6]). Let `Xs` be a tuple of `n` random variables over `A` and `X` be a random variable over `n.-tuple A`. `sum_n Xs X` holds when `X` is the sum of `Xs`. We also specialize this definition to the sum of independent random variables. Equipped with above definitions, we derive the standard properties of the expected value, such as its linearity, but also properties of the variance. See [14] for details.

2.3 The Weak Law of Large Numbers

The weak law of large numbers is the first fundamental theorem of probability. Intuitively, it says that the average of the results obtained by repeating an experiment a large number of times is close to the expected value. Formally, let `Xs` be a tuple of `n` *identically distributed* random variables, i.e., random variables with the same distribution `P`. Let us assume that these random variables are independent and let us write `X` for their sum, μ for their common expected value, and σ^2 for their common variance. The weak law of large numbers says that the outcome of the average random variable `avg_rv X` gets closer to μ:

```
Lemma wlln ε : 0 < ε →
  Pr p_X [pred x | Rabs (avg_rv X x - μ) ≥_R ε] ≤
    σ^2 / (n+1 * ε ^ 2).
```

See [14] for the proof of this lemma using the Chebyshev inequality.

3 Entropy and Typical Sequences

We formalize the central concept of a typical sequence. Intuitively, a typical sequence is an n-tuple of symbols (where n is large) that is expected to be observed. For example, a tuple produced by a binary source that emits 0's with probability 2/3 is typical when it contains approximately two thirds of 0's. The precise definition of typical sequences requires the definition of the entropy and their properties relies on a technical result known as the Asymptotic Equipartition Property. (One can find an alternative HOL version of most definitions and properties in this section in [13].)

3.1 Entropy and Asymptotic Equipartition Property

We define the entropy of a random variable with distribution `P` over `A` as follows (where log is the binary logarithm, derived from the standard library of Coq):

```
Definition H := - Σ_(i ∈ A) P i * log (P i).
```

The Asymptotic Equipartition Property (AEP) is a property about the outcome of several random variables that are independent and identically distributed (i.i.d.). Let us assume an `n`-tuple of i.i.d. random variables with distribution `P` over `A`. The probability of the outcome `x` (of type `n.-tuple A`) is:

```
Definition Ptuple x := Π_(i < n) P x_i.
```

(The big product operator comes from SSReflect, `x_i` is for accessing the `i`th element of the tuple `x`.) Informally, the AEP states that, in terms of probability, `- (1 / n) * log(Ptuple P x)` is "close to" the entropy `H P`. Here, "close to" means that, given an ε `> 0`, the probability that `- (1 / n) * log(Ptuple P x)` and `H P` differ by more than ε is less than ε, for `n` greater than the bound `aep_bound` ε defines as follows:

```
Definition aep_σ^2 := Σ_(x ∈ A) P x * (log (P x))^2 - (H P)^2.
Definition aep_bound ε := aep_σ^2 P / ε^3.
```

The probability in the AEP is taken over a *tuple distribution*. Given a distribution `P` over `A`, the tuple distribution `P^n` over `n.-tuple A` is defined as follows:

```
Definition Ptuple_dist : dist [finType of n.-tuple A].
apply mkDist with Ptuple. ... Defined.
```

Using above definitions, the AEP can now be stated formally. Its proof is an application of the weak law of large numbers (Sect. 2.3):

```
Lemma aep : aep_bound P ε ≤ n+1 →
  Pr (P^n+1) [pred x | 0 <_R P^n+1 x ∧
    Rabs (- (1 / n+1) * log (Ptuple P x) - H P) ≥_R ε ] ≤ ε.
```

3.2 Typical Sequences: Definition and Properties

Given a distribution `P` over `A` and an ε, a typical sequence is an `n`-tuple with probability "close to" 2^{-nHP}:

```
Definition typ_seq (x : n.-tuple A) ε :=
 exp (- n * (H P + ε)) <R= Ptuple P x <R= exp (- n * (H P - ε)).
```

Let us note $\mathcal{TS}$ the set of typical sequences. Using the AEP, we prove that the probability to observe a typical sequence for large `n` is close to 1, corresponding to the intuition that it is expected to be observed in the long run:

```
Lemma Pr_TS_1 : aep_bound P ε ≤ n+1 →
  Pr (P^n+1) [pred i ∈ TS P n+1 ε] ≥ 1 - ε.
```

The cardinal of $\mathcal{TS}$ is nearly 2^{nHP}. Precisely, it is upper-bounded by $2^{n(HP+\varepsilon)}$, and lower-bounded by $(1-\varepsilon)2^{n(HP-\varepsilon)}$ for n big enough:

```
Lemma TS_sup : | TS P n ε | ≤ exp (n * (H P + ε)).
Lemma TS_inf : aep_bound P ε ≤ n+1 →
  (1 - ε) * exp (n+1 * (H P - ε)) ≤ | TS P n+1 ε |.
```

4 The Source Coding Theorem

The source coding theorem (a.k.a. the noiseless coding theorem) is a theorem for data compression. The basic idea is to replace frequent words with alphabet sequences and other words with a special symbol. Let us illustrate this with an example. The combination of two Roman alphabet letters consists of 676 ($= 26^2$) words. Since $2^9 < 676 < 2^{10}$, 10 bits are required to represent all the words. However, by focusing on often-used English words ("as", "in", "of", etc.), we can encode them with less than 9 bits. Since this method does not encode rarely-used words (such as "pz") decoding errors can happen. Given an information source known as a discrete memoryless source (DMS) that emits all symbols with the same distribution `P`, the source coding theorem gives a theoretical lower-bound (namely, the entropy `H P`) for compression rates for compression with negligible error-rate.

4.1 Definition of a Source Code

Given a set `A` of symbols, a `k,n`-source code is a pair of an encoder and a decoder. The encoder maps a `k`-tuple of symbols to an `n`-tuple of bits and the decoder performs the corresponding decoding operation:

```
Definition encT := k.-tuple A → n.-tuple bool.
Definition decT := n.-tuple bool → k.-tuple A.
Record scode := mkScode { enc : encT ; dec : decT }.
```

The rate of a `k,n`-source code `sc` is defined as the ratio of bits per symbol:

```
Definition SrcRate (sc : scode) := n / k.
```

Given a DMS with distribution `P` over `A`, the error rate of a source code `sc` (notation: $\bar{e}_{src}$`(P , sc)`) is defined as the probability of failure for the decoding of encoded sequences:

```
Definition SrcCodeErrRate :=
  Pr (P^k) [pred x | dec sc (enc sc x) ≠ x].
```

4.2 Source Coding Theorem—Direct Part

Given a source of symbols from the alphabet `A` with distribution `P`, there exist source codes of rate `r` $\in \mathcal{Q}^+$ (the positive rationals) larger than the entropy `H P` such that the error rate can be made arbitrarily small:

```
Theorem source_coding_direct : ∀ λ, 0 < λ < 1 →
  ∀ r : 𝒬⁺, H P < r ≤ 1 →
    ∃ k, ∃ n, ∃ sc : scode A k n,
      r = SrcRate sc ∧ ē_src(P , sc) ≤ λ.
```

Source Coding using the Typical Set. The crux of the proof is to instantiate with an adequate source code. We first define the corresponding encoder and decoder functions. For a set `S` of `k+1`-tuples, the encoder `f` encodes the ith element of `S` as the binary encoding of $i+1$ and elements not in `S` as a string of 0's:

```
Definition f : encT A k+1 n := fun x ⇒
 if x ∈ S then
  let i := index x (enum S) in Tuple (size_nat2bin_b i+1 n)
 else
  [tuple of nseq n false].
```

(`enum S` is the lists of all the elements of `S`; `index` returns the index of an element in a list; `nat2bin_b` is a function that converts an integer $i < 2^n$ to a bitstring, `size_nat2bin_b` being the proof that this bitstring has length n.)

The definition of the decoder requires to have a default element `def` $\in$ `S`. The decoder ϕ returns the $i-1$th element of `S` if i is smaller than the cardinal of `S`, and some default value from `S` otherwise:

```
Definition ϕ : decT A k+1 n := fun x ⇒
  let i := tuple2N x in
  if i is 0 then def else
    if i-1 < | S | then nth def (enum S) i-1 else def.
```

(`tuple2N` interprets bitstrings as Peano integers; `nth` picks up the nth element of a list.) By construction, `f` and ϕ perform lossless coding:

```
Lemma ϕ_f i : ϕ (f i) = i ↔ i ∈ S.
```

In the proof of the source coding theorem, the set `S` is actually taken to be the set $\mathcal{TS}$ of typical sequences and there exists a default element `def` $\in \mathcal{TS}$ when `k` is big enough, bound to be made more precise below.

Formalization of the Bounds. Above, we explained how to construct the required source code. Technically, in the formal proof, it is also important to correctly instantiate `n` and `k` (given the source rate `r`, λ and the distribution `P`), such that `k` is "big enough" for the lemma `ϕ_f` to hold. This aspect of the proof is usually overlooked in the information theory literature, so that the (precise) formal definition of these bounds is one of our contributions.

Let us define the following quantities:

```
Definition ε := Rmin (r - H P) λ.
Definition δ := Rmax (aep_bound P (ε / 2)) (2 / ε).
```

`k` must satisfy $\delta \leq$`k` and `k * r` must be a natural. Such a `k` can be constructed using the following lemma:

```
Lemma SrcDirectBound n d m : 0 < m →
  { k | m ≤ (k+1 * d+1) ∧
        frac_part ((k+1 * d+1) * (n / d+1)) = 0}.
```

Let us assume that the rate is `r = num / den+1`. If we note `k'` the natural constructed via the above lemma by taking `n` to be the numerator `num`, `d` to be `den`, and `m` to be δ, then it is sufficient to take `n` equal to `k'+1 * num` and `k` equal to `k'+1 * den+1`.

At this point, we have thoroughly explained how to instantiate the source code required by the source coding theorem. The proof is completed by appealing to the properties of typical sequences (in particular, lemmas `Pr_𝒯𝒮_1` and `𝒯𝒮_sup` from Sect. 3.2). The successive steps of the proof can be found in [14].

4.3 Source Coding Theorem—Converse Part

The converse of the Shannon's source coding theorem shows that any source code whose rate is smaller than the entropy of a source with distribution `P` over `A` has non-negligible error-rate:

```
Theorem source_coding_converse : ∀ λ, 0 < λ < 1 →
  ∀ r : 𝒬⁺, 0 < r < H P →
    ∀ n k (sc : scode A k+1 n),
      r = SrcRate sc →
      SrcConverseBound P (num r) (den r) n λ ≤ k+1 →
     ē_src(sc , P) ≥ λ.
```

where the bound `SrcConverseBound` gives a precise meaning to the claim that would otherwise be informally summarized as "for `k` big enough":

```
Definition ε := Rmin ((1 - λ) / 2) ((H P - r) / 2).
Definition δ := Rmin ((H P - r) / 2) (ε / 2).
Definition SrcConverseBound := Rmax (Rmax
  (aep_bound P δ) (- ((log δ) / (H P - r - δ)))) (n / r).
```

The proof of the converse part of the source coding theorem is a bit simpler than the direct part because no source code needs to be constructed. See [14] for the detail of the proof steps.

5 Formalization of Channels

5.1 Discrete Memoryless Channel

A *discrete channel* with input alphabet X and output alphabet Y is a (probability transition) matrix (ptm) that expresses the probability of observing an output symbol given some input symbol; it associates to each input a distribution of the corresponding outputs (as ensured by the proofs ptm0 and ptm1):

```
Record W := mkW {
  ptm :> Y → X → R ;
  ptm0 : ∀ y x, 0 ≤ ptm y x ;
  ptm1 : ∀ x, Σ_(y ∈ Y) ptm y x = 1 }.
```

The *nth extension of a discrete channel* is the generalization of a discrete channel to the communication of several symbols:

```
Record Wn n := mkWn {
  nptm :> {: n.-tuple Y} → {: n.-tuple X} → R ;
  nptm0 : ∀ (y : n.-tuple _) x, 0 ≤ nptm y x ;
  nptm1 : ∀ x, Σ_(y ∈ {: n.-tuple Y}) nptm y x = 1 }.
```

A *discrete memoryless channel* (DMC) models channels whose inputs do not depend on past outputs. It is the special case of the nth extension of a discrete channel defined as follows (again, we omit the proofs for nptm0 and nptm1):

```
Definition DMC (w : W) n : Wn n.
apply mkWn with (fun y x ⇒ Π_(i < n) w y_i x_i). ... Defined.
```

5.2 Mutual Information and Channel Capacity

Given a discrete channel w with input alphabet X and output alphabet Y, and an input distribution P, there are two important distributions: the output distribution and the mutual distribution. The *output distribution* (notation: d(P , w)) is the distribution of the outputs:

```
Definition out_dist (P : dist X) (w : W Y X) : dist Y.
apply mkDist with (fun y ⇒ Σ_(x ∈ X) w y x * P x). ...Defined.
```

The *mutual distribution* (notation: d(P ; w)) is the joint distribution of the inputs and the outputs:

```
Definition mut_dist (P : dist X) (w : W Y X) :
  dist ([finType of X * Y]).
apply mkDist with (fun xy ⇒ w xy.2 xy.1 * P xy.1). ...Defined.
```

The *output entropy* (resp. *mutual entropy*) is the entropy of the output distribution (resp. mutual distribution), hereafter noted H(P , w) (resp. H(P ; w)).

The *mutual information* (notation: I(P ; w)) is a measure of the amount of information that the output distribution contains about the input distribution:

```
Definition mut_info_W (P : dist X) (w : W Y X) :=
  H P + H(P , w) - H(P ; w).
```

Finally, the *information channel capacity* is defined as the least upper bound of the mutual information taken over all possible input distributions:

```
Definition upper_bound {A} (f : A → R) b := ∀ a, f a ≤ b.
Definition lub {A} (f : A → R) b :=
  upper_bound f b ∧ ∀ b', upper_bound f b' → b ≤ b'.
Definition capacity (w : W Y X) c := lub (fun P ⇒ I(P ; w)) c.
```

It may not be immediate why the supremum of the mutual information is called capacity. The goal of the channel coding theorem is to ensure that we can distinguish between two outputs (actually sets of outputs because of potential noise), so as to be able to deduce the corresponding inputs without ambiguity. For each input (of n symbols), there are approximately $2^{n(H(P;w)-HP)}$ typical outputs because $H(P;w) - HP$ is the entropy of the output knowing the input. On the other hand, the total number of typical outputs is approximately $2^{nH(P,w)}$. Since this set has to be divided into sets of size $2^{n(H(P;w)-HP)}$, the total number of disjoint sets is less than or equal to $2^{n(H(P,w)-(H(P;w)-HP))} = 2^{nI(P;w)}$.

5.3 Example: The Binary Symmetric Channel

We illustrate above definitions with the simplest model of channel with errors: the `p`-binary symmetric channel (`BSC` below). In such a channel, the input and output symbols are taken from the same alphabet `X` with only two symbols (hypothesis noted `HX` below). Upon transmission, the input is flipped with probability `p` (with hypothesis `Hp : 0 < p < 1`):

```
Definition BSC : W X X.
apply mkW with (fun y x ⇒ if x = y then 1 - p else p).
... Defined.
```

For convenience, we introduce the *binary entropy function*:

```
Definition H₂ p := - p * log p - (1 - p) * log (1 - p).
```

For any input distribution `P`, we prove that the mutual information can actually be expressed by only the entropy of the output distribution and the binary entropy function:

```
Lemma IPW : I(P ; BSC HX Hp) = H(P , BSC HX Hp) - H₂ p.
```

The maximum of the binary entropy function on the interval $(0, 1)$ is 1, fact that we proved formally in Coq by appealing to the standard library for reals[1]:

```
Lemma H₂_max : ∀ q, 0 < q < 1 → H₂ q ≤ 1.
```

This fact gives an upper-bound for the entropy of the output distribution:

```
Lemma H_out_dist_max : H(P , BSC HX Hp) ≤ 1.
```

The latter bound is actually reached for the uniform input distribution:

[1] Modulo a slight extension of the corollary of the mean value theorem to handle derivability of partial functions.

```
Definition binary_uniform : dist X.
apply mkDist with (fun x ⇒ 1 / 2). ... Defined.
Lemma H_binary_uniform : H(binary_uniform , BSC HX Hp) = 1.
```

Above facts imply that the capacity of the p-binary symmetric channel can be expressed by a simple closed formula:

```
Theorem BSC_capacity : capacity (BSC HX Hp) (1 - H₂ p).
```

5.4 Jointly Typical Sequences

Let us consider a channel w with input alphabet X, output alphabet Y, and input distribution P. A *jointly typical sequence* is a pair of two sequences such that: (1) the first sequence is typical for P, (2) the second sequence is typical for the output distribution d(P , w), and (3) the pair is typical for the mutual distribution d(P ; w)[2]:

```
Definition jtyp_seq n (xy : n.-tuple (X * Y)) ε :=
  typ_seq P ε (uzip1 xy) ∧
  typ_seq (d(P , w)) ε (uzip2 xy) ∧
  typ_seq (d(P ; w)) ε xy.
```

We note $\mathcal{JTS}$ the set of jointly typical sequences. The number of jointly typical sequence is upper-bounded by $2^{n(H(P;w)+\varepsilon)}$:

```
Lemma 𝒥𝒯𝒮_sup ε : | 𝒥𝒯𝒮 P w n ε| ≤ exp (n * (H(P ; w) + ε)).
```

Now follow two lemmas that will be key to prove the channel coding theorem. With high probability (probability taken over the tuple distribution of the mutual distribution), the sent input and the received output are jointly typical:

```
Lemma 𝒥𝒯𝒮_1 : 𝒥𝒯𝒮_1_bound ≤ n →
  Pr ((d( P ; w)^n)) [pred x ∈ 𝒥𝒯𝒮 P w n ε] ≥ 1 - ε.
```

The bound $\mathcal{JTS}$_1_bound is defined as follows:

```
Definition 𝒥𝒯𝒮_1_bound :=
  maxn (up (aep_bound P (ε/3)))
 (maxn (up (aep_bound (d(P , w)) (ε/3)))
       (up (aep_bound (d(P ; w)) (ε/3)))).
```

(up r is the ceiling of r, this is a function from the Coq standard library.) This bound will later appear again in the proof of the channel coding theorem (Sect. 6.3).

In contrast, the probability of the same event (joint typicality) taken over the product distribution of the inputs and the outputs considered independently tends to 0 as n gets large:

```
Lemma non_typical_sequences : Pr ((P^n) × ((d(P , w))^n))
  [pred x ∈ 𝒥𝒯𝒮 P w n ε] ≤ exp (- n * (I( P ; w) - 3 * ε)).
```

[2] Informal definitions about jointly typical sequences seeminglessly switch between $(X \times Y)^n$ and $X^n \times Y^n$; this translates formally to projections and casts that we do not represent explicitly in this paper.

6 The Channel Coding Theorem

6.1 Formalization of a Channel Code

The purpose of a code is to transform the input of a channel (typically, by adding some form of redundancy) so that the transmitted information can be recovered correctly from the output despite of potential noise. Concretely, given input alphabet `X` and output alphabet `Y`, a (channel) code is (1) a set `M` of codewords, (2) an encoding function that turns a codeword into `n` input symbols, and (3) a decoding function that turns `n` output symbols back into the original codeword (or possibly fails):

```
Definition encT := {ffun M → n.-tuple X}.
Definition decT := {ffun n.-tuple Y → option M}.
Record code := mkCode { enc : encT ; dec : decT }.
```

The *rate* of a code is defined as follows:

```
Definition CodeRate (c : code) := log (| M |) / n.
```

For convenience, we introduce the following predicate to characterize (channel) code rates:

```
Definition CodeRateType r := ∃ n, ∃ d,
  0 < n ∧ 0 < d ∧ r = log n / d.
```

We now define the error-rate. For this purpose, we introduce a new kind of distribution. Given a channel `w` and a tuple of inputs `x`, we define (and note "`w (| x )`") the distribution of outputs knowing that `x` was sent:

```
Definition DMC_cond (w : W) n (x : n.-tuple X) :
  dist [finType of n.-tuple Y].
apply mkDist with (fun y ⇒ (DMC w n) y x). ... Defined.
```

Using this distribution, we first define the probability of decoding error knowing that the codeword `m` from the code `c` was sent (notation: `e(w , c) m`):

```
Definition e (w : W Y X) c m :=
  Pr (w (| enc c m) ) [pred y | dec c y ≠ Some m].
```

Finally, we define the error rate as the average probability of error for a code `c` over channel `w` (notation: $\bar{e}_{cha}$`(w , c)`):

```
Definition ChanCodeErrRate := 1 / | M | * Σ_(m ∈ M) e(w, c) m.
```

6.2 Channel Coding Theorem—Statement of the Direct Part

The (noisy-)channel coding theorem (a.k.a. Shannon's theorem) is a theorem for reliable information transmission over a noisy channel. The basic idea is to represent the original message by a longer message. Let us illustrate this with an example. Assume the original message is either 0 or 1 and is sent over a p-binary symmetric channel (see Sect. 5.3). The receiver obtains the wrong message with probability p. Let us now consider that the original message is 0 and encode 0 into 000 before transmission (in other words, we use a repetition encoding with code rate 1/3). The receiver obtains a message from $\{000, 001, 010, 100\}$ with probability $(1-p)^3 + 3p(1-p)^2$ and it guesses the original message 0 by majority vote. The error probability $1 - ((1-p)^3 + 3p(1-p)^2)$ is smaller than p.

One may guess that the smaller the code rate is, the smaller the error probability becomes. Given a discrete channel `w` (with input alphabet `X` and output alphabet `Y`), the channel coding theorem guarantees the existence of an encoding function and a decoding function such that the code rate is not small (but smaller than the capacity `cap`—hypothesis `capacity w cap`) but is with negligible error-rate:

```
Theorem channel_coding r : CodeRateType r → r < cap →
  ∀ ε, 0 < ε →
    ∃ n, ∃ M, ∃ c : code X Y M n,
      r = CodeRate c ∧ ē_cha(w, c) < ε.
```

6.3 Channel Coding Theorem—Proof of the Direct Part

We formalize a proof by "random coding". In a nutshell: we first fix the decoding function and then select an appropriate encoding function by checking all the possible ones. Selection operates using a criterion about the average error-rate of all the possible encoding functions, weighted according to a well-chosen distribution.

Decoding by Joint Typicality. We first fix the decoding function `jtdec`. Given the channel output `y`, `jtdec` looks for a codeword `m` such that the channel input `f m` is jointly typical with `y`. If a unique such codeword is found, it is declared to be the sent codeword (`[pick m | P m]` is a SSReflect construct that picks up an element `m` satisfying the predicate `P`):

```
Definition jtdec P w ε (f : encT X M n) : decT Y M n :=
  [ffun y ⇒ [pick m |
    ((f m, y) ∈ 𝒥𝒯𝒮 P w n ε) ∧
    (∀ m', (m' ≠ m) ⇒ ((f m', y) ∉ 𝒥𝒯𝒮 P w n ε))]].
```

Criterion for Encoder Selection. We are looking for a code such that the error-rate can be made arbitrarily small. The following lemma provides a sufficient condition for the existence of such a code:

```
Lemma good_code_sufficient_condition (P : dist X) w ε
    (ϕ : encT X M n → decT Y M n) :
 Σ_(f : encT X M n) (wght P f * ē_cha(w , mkCode f (ϕ f))) < ε →
 ∃ f, ē_cha(w , mkCode f (ϕ f)) < ε.
```

where `wght` is the distribution of encoding functions defined as follows:

```
Definition wght (P : dist X) : dist [finType of (encT X M n)].
apply mkDist with
 (fun f : encT X M n ⇒ Π_(m ∈ M) Ptuple P (f m)). ... Defined.
```

The Main Lemma. Our theorem can be derived from the following technical lemma by just proving the existence of appropriate ε_0 and `n`. This lemma establishes that there exists a set of codewords `M` such that decoding by joint typicality meets the above criterion:

```
0 Lemma random_coding_good_code : ∀ ε, 0 ≤ ε →
1   ∀ r, CodeRateType r →
2     ∀ ε0, ε0_condition r ε ε0 →
3       ∀ n, n_condition r ε0 n →
4   ∃ M : finType , 0 < |M| ∧ |M| = Int_part (exp (n * r)) ∧
5   let Jtdec := jtdec P w ε0 in
6   Σ_(f : encT X M n) (wght P f * ē_cha(w , mkCode f (Jtdec f))) < ε.
```

In this lemma, the fact that the rate `r` is bounded by the mutual information appears in the condition `ε0_condition`:

```
Definition ε0_condition r e e0 :=
  0 < e0 ∧ e0 < e / 2 ∧ e0 < (I(P ; w) - r) / 4.
```

The condition `n_condition` corresponds to the formalization of the restriction "for `n` big enough" (we saw the bound `𝒥𝒯𝒮_1_bound` in Sect. 5.4):

```
Definition n_condition r e0 n := 0 < n ∧ - log e0 / e0 < n ∧
  frac_part (exp (n * r)) = 0 ∧ 𝒥𝒯𝒮_1_bound P w e0 ≤ n.
```

Proof of the Main Lemma. The first thing to observe is that by construction the error-rate averaged over all possible encoders does not depend on which codeword `m` was sent:

```
Lemma error_rate_symmetry (P : dist X) (w : W Y X) ε :
 0 ≤ ε → let Jtdec := jtdec P w ε in
  ∀ m m',
 Σ_(f : encT X M n) (wght P f * e(w, mkCode f (Jtdec f)) m) =
 Σ_(f : encT X M n) (wght P f * e(w, mkCode f (Jtdec f)) m').
```

Therefore, the left-handside of the conclusion of the main lemma (line 6 above) can be rewritten by assuming that the codeword `0` was sent:

```
Σ_(f : encT X M n)
  wght P f * Pr (w (|f 0)) [pred y ∈ not_preimg (Jtdec f) 0]
```

where `not_preimg (Jtdec f) 0` is the set of outputs that do not decode to `0`.

Let us write `ℰ f m` for the set of outputs `y` such that `(f m, y) ∈ 𝒥𝒯𝒮 P w n ε`. Assuming that `0` was sent, a decoding error occurs when (1) the input and the output are not jointly typical, or (2) when a wrong input is jointly typical with the output (`~:` is a notation for set complementation):

```
[set x ∈ not_preimg (JTdec f) 0] =i
  (~: ℰ f 0) ∪ ⋃_(i : M | i ≠ 0) ℰ f i.
```

Using the fact that the probability of a union is smaller that the sum of the probabilities, the left-handside of the conclusion of the main lemma can be bounded by the following expression:

```
Σ_(f : encT X M n)
   wght P f * Pr (w (|f 0)) [pred y ∈ ~: ℰ f 0] +     (* (1) *)
Σ_(i|i ≠ 0)Σ_(f : encT X M n)
   wght P f * Pr (w (|f 0)) [pred y ∈ ℰ f i]          (* (2) *)
```

The first summand (1) can be rewritten into

```
Pr (d(P ; w)^n) [pred y ∈ ~: 𝒥𝒯𝒮 P w n ε₀]
```

which can be bounded using the lemma `𝒥𝒯𝒮_1` (Sect. 5.4). The second summand (2) can be rewritten into

```
k * Pr (P^n × (d(P, w))^n) [pred x ∈ 𝒥𝒯𝒮 P w n ε₀]
```

which can be bounded using the lemma `non_typical_sequences` (Sect. 5.4). The bounds ε_0 and `n` have been carefully chosen so that the proof can be concluded with symbolic manipulations. See [14] for details.

7 Related Work

The formalization of Shannon's theorems in this paper as well as the formalization of advanced information-theoretic concepts (channels, jointly typical sequences, etc.) are new. Yet, one can find formalization of more basic concepts of information theory in the literature. [9] formalizes (conditional) entropy and (conditional) mutual information (based on the seminal work by Hurd [4]), defines a notion of information leakage, and applies it to the verification of privacy properties of a protocol. [13] provides a formalization of the AEP and presents the source coding theorem as a potential application; in other words, our paper can be seen as the direct continuation of [13], though in a different proof-assistant.

For the purpose of this paper, we formalized finite probability using SSReflect. As we have hinted at several times in this paper, this formalization was important to take advantage of SSReflect's library (in particular, canonical big operators [7]). We limit ourselves to finite probability because it is enough for our purpose (as for the information theory formalized in [9]). [8] provides an alternative formalization of probabilities in Coq but that is biased towards verification of randomized algorithms. Hasan et al. formalize probability theory on more general grounds in the HOL proof-assistant: [6] formalizes the expectation properties (this is also based on the work by Hurd [4]), [11] provides a formalization of the Chebyshev inequality and of the Weak Law of Large Numbers.

Our formalization of the source coding theorem follows [3, Chapter 1] with nevertheless much clarification (in particular, formalization of bounds).

8 Conclusion and Future Work

We presented a formalization of information-theoretic definitions and lemmas in the SSReflect extension of the Coq proof-assistant. Besides basic material such as finite probability and typical sequences, this formalization includes a formalization of channels (duly illustrated with the example of the binary symmetric channel), codes (for source and channel coding), and jointly typical sequences. We use this formalization to produce the first formal proofs of the source coding theorem (direct and converse parts), that establishes the limit to possible data compression, and the direct part of the channel coding theorem, that establishes the limit to reliable data transmission over a noisy channel. Compared to pencil-and-paper proofs, our formalization has the added value to make precise the construction of asymptotic bounds.

We believe that the library that we have formalized can be used to formalize further results about information theory (primarily, the converse of the channel coding theorem) and also results of unconditional security (e.g., the proof of the perfect secrecy of the one-time pad [2]).

The channel coding theorem proves the existence of codes for reliable data transmission. Such codes play a critical role in IT products (e.g., LDPC codes in storage devices). As a first step towards the verification of the implementation of codes, we have been working on formalizing their basic properties ([14] already provides several standard proofs about coding theory).

References

1. Shannon, C.E.: A Mathematical Theory of Communication. Bell System Technical Journal 27, 379–423, 623–656 (1948)
2. Shannon, C.E.: Communication Theory of Secrecy Systems. Bell System Technical Journal 28, 656–715 (1949)
3. Uyematsu, T.: Modern Shannon Theory, Information theory with types. Baifukan (1998) (in Japanese)
4. Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD Thesis, Trinity College, University of Cambridge, UK (2001)
5. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. Wiley-Interscience (2006)
6. Hasan, O., Tahar, S.: Verification of Expectation Using Theorem Proving to Verify Expectation and Variance for Discrete Random Variables. J. Autom. Reasoning 41, 295–323 (2008)
7. 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)
8. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in COQ. Sci. Comput. Program. 74(8), 568–589 (2009)
9. Coble, A.R.: Anonymity, Information, and Machine-Assisted Proof. PhD Thesis, King's College, University of Cambridge, UK (2010)
10. The COQ Development Team. Reference Manual. Version 8.3. INRIA (2004-2010), http://coq.inria.fr
11. 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)

12. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Version 10. Technical report RR-6455. INRIA (2011)
13. Mhamdi, T., Hasan, O., Tahar, S.: Formalization of Entropy Measures in HOL. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 233–248. Springer, Heidelberg (2011)
14. Affeldt, R., Hagiwara, M.: Formalization of Shannon's Theorems in SSReflect-COQ. COQ scripts, `http://staff.aist.go.jp/reynald.affeldt/shannon`

Stop When You Are Almost-Full
Adventures in Constructive Termination

Dimitrios Vytiniotis[1], Thierry Coquand[2], and David Wahlstedt[2]

[1] Microsoft Research, Cambridge
dimitris@microsoft.com
[2] University of Gothenburg
coquand@chalmers.se, david.wahlstedt@gmail.com

Abstract. Disjunctive well-foundedness, size-change termination, and well-quasi-orders are examples of techniques that have been successfully applied to program termination. Although these works originate in different communities, they rely on closely related principles and both employ similar arguments from Ramsey theory. At the same time there is a notable absence of these techniques in programming systems based on constructive type theory. In this paper we'd like to highlight the aforementioned connection and make the core ideas widely accessible to theoreticians and programmers, by offering a development in type theory which culminates in some novel tools for induction. Inevitably, we have to present some Ramsey-like arguments: Though similar proofs are typically classical, we offer an entirely constructive development based on the work of Bezem and Veldman, and Richman and Stolzenberg.

1 Introduction

Program termination has always been an exciting subject, dating back to the early days of computing. The reason is because program termination is at the same time *important* for software reliability, and *difficult* for general classes of programs. Despite the difficulties, however, several research communities have managed to make good progress.

Over the recent years, the *transition invariants* [26] method has been an extremely successful approach for automatic proofs of program termination, leading to industrial-strength tools [9]. *Size-change termination* (SCT) [21,16,30] is another very successful recent method, though similar techniques date back to the early 90's [28]. Both lines of work rely on formal arguments from Ramsey theory [15], first introduced in the termination literature in [14] and also used in [12]. Furthermore, research on online termination testing [22] and supercompilation [31] has for a while been using termination criteria for function reductions and inlining based on *well-quasi-orders*, often employing Ramsey-like arguments to form more complex termination testing criteria from simpler ones.

There is an intimate connection between these worlds, and a notable absence of similar techniques to help programmers *prove* the totality of their definitions in programming systems based on constructive type theory. To quote some related work on size-change termination for Isabelle [18]:

"Our proof uses classical logic, including the (infinite, but countable) axiom of choice. It would be interesting to investigate if the proof can be modified to work in a weaker framework"

We show that this is indeed possible. We reveal the connection between the aforementioned previous works, and make the core ideas widely accessible to theoreticians and programmers, by offering a development which introduces some novel variations of induction principles. Inevitably, we have to present some Ramsey-like arguments, proved *constructively*, in the footsteps of Bezem and Veldman [35], and Richman and Stolzenberg [27]. Specifically, our contributions with this paper are:

– We introduce a novel tool for proving termination, that of *almost-full* relations (Section 2), which is a weaker version of the more traditional well-quasi-orders, originating in intuitionistic mathematics.
– We formally explain the connection between almost-full relations and well-founded relations (Section 3), and prove a new induction theorem based on almost-full relations. (Section 3.1)
– We demonstrate that almost-full relations compose nicely to form other almost-full relations (Section 4), yielding intuitive proof obligations for termination which involve relation inclusion lemmas instead of accessibility predicates. In this context, we prove and use an intuitionistic version of Ramsey's theorem for binary relations.
– We can use our method to show complex examples from SCT (Section 5). We show that the SCT principle can be intuitionistically proved from our induction principle. We show that this is also the case for the Terminator rule (based on the so-called *disjunctive well-foundedness*). (Section 6)
– We generalize our statement of the intuitionistic Ramsey theorem to relations of transfinite arities, and offer an elegant and simpler proof of this theorem than older attempts [10,13].(Section 7)

Our accompanying Coq development does not make use of any "non-standard" axioms (such as classical facts). In addition to the Coq formalization, we've also produced an Agda formalization of Section 7.[1]

The new induction principles proposed in this paper are not-necessarily more expressive or easier to use than other (particularly recent [8,17,33]) related work – this is a topic that deserves more engineering and automation support. On the other hand, our new induction principles are quite amenable to the same automation that made Terminator and SCT successful, thanks to the composability of almost-full relations and the nature of the user obligations that arise.

Apart from contributing to the large arsenal of techniques for recursion [4,7,23,33,8], the other significant contribution of this article is to bring together ideas from different research communities in a type-theoretic framework.

[1] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.IntuitionisticRamseyTheorem`

2 Well-Quasi-Orders and Almost-Full Relations

Our starting point will be the well-known notion of a *well-quasi-order* (WQO):

Definition 1 (Well-quasi-order). *A binary relation $\preceq$ on a set* `X` *is a well quasi order if (i) it is transitive and (ii) for every infinite sequence s of elements of* `X` *there exist i and j with $i < j$ such that $s_i \preceq s_j$.*

For example, on the type `nat`, the relation $\leq$ (`le` in Coq) is such a WQO.

A great use of WQOs is for *online termination testing*: Assume that the observed state of a program forms a sequence of values $s_1, s_2, \ldots$ – online termination testing aims to detect if that sequence s could be infinite by examining a finite prefix of s. Consider the following online termination tester which accepts a user-provided WQO $\preceq$ as input: We keep a record of all values we have observed already and every time a new value s_{new} appears, we check if for some old value s_{old} it is $s_{old} \preceq s_{new}$. If this is true then we raise an error, otherwise we record s_{new} in our history and wait for the next value. Now, if the sequence was infinite then we will definitely raise an error at some point (because $\preceq$ is a WQO). Of course, conservatively, we might raise an error even when the sequence is not infinite because the WQO provided was too conservative.

The merits of WQOs for online termination testing have been discussed in previous work [22] so we will not go into details here. Their main advantage is that they can form extremely lenient termination tests by combining simpler WQOs (at the cost of having to record big portions of history).

2.1 Almost-Full Relations

The mechanism described above is by now well-established for online termination testing, so it is quite natural to ask how it would look in type theory and check if it can be used to *prove* termination, in addition to testing for termination.

Surprisingly, it turns out that relations that satisfy property (ii) in the definition of WQOs have been proposed by mathematicians in an entirely different domain: the development of an intuitionistic version of Ramsey theory [35]. These are the *almost-full* (AF) relations (term coined by Wim Veldman), and for the rest of this paper we will focus on *binary* AF relations. Bezem and Veldman used condition (ii) as the defining condition of AF relations (see also [34]). They additionally postulated the axiom of bar induction to develop an intuitionistic proof of Ramsey's theorem. We give below a *direct inductive characterization* of AF relations:

```
Inductive almost_full X : (X → X → Prop) → Prop :=
| AF_ZT : ∀ (R : X → X → Prop), (∀  x y, R x y) → almost_full R
| AF_SUP : ∀ R,
   (∀  x, almost_full (fun y z ⇒ R y z ∨ R x y)) → almost_full R.
```

Our goal with this definition is to characterize relations that eventually "go up": if we repeatedly request elements from an opponent, these elements are guaranteed eventually to "go up" from some element we have previously encountered in the sequence. Concretely, assume that we know that a relation `R` satisfies `almost_full R`. If the proof object is `AF_ZT` then we know that *any* two elements in an infinite sequence will be related. If on the other hand the proof object is built from the `AF_SUP` constructor then, if we receive a first element `x` in a sequence we know that `almost_full (fun y z ⇒ R y z ∨ R x y)`. This means that in any infinite sequence that starts with `x`, either there exist two elements in the rest of the sequence related by `R`, or some element which is related to the first element `x`.

In fact we can prove condition (ii) of the definition of WQOs:

```
Corollary af_inf_chain (X : Set) (R : X → X → Prop):
   almost_full R → ∀ (f : nat → X), ∃ m, ∃ n, (m < n) ∧ R (f m) (f n).
```

It is interesting to observe that corollary `af_inf_chain` is quite analogous to the "no infinite descending chain" property which can be proved intuitionistically from Coq's inductive definition of well-founded relations (based on accessibility predicates) [4]. The converse of `af_inf_chain` holds only classically (paragraph 8.6.1 of [35] suggests that intuitionistically there exists a counterexample, unless intuitionistic type theory is inconsistent with Church's thesis). This makes it difficult to use property (ii) as the very defining property of AF relations (instead of our inductive characterization) because that alternative definition cannot be used for induction (see Theorem `wf_from_af` in Section 3.1).

Notice also that – perhaps surprisingly – we have ignored the transitivity condition (i) of WQOs, an issue that we return to in Section 4.2.

3 Well-Founded vs. Almost-Full Relations

To build up some more intuitions about AF relations, we now turn to the connection between AF relations and well-founded relations. A well-founded relation can be constructively characterized as a relation where every element in its domain is *accessible*. The corresponding (standard) Coq definitions are:

```
Inductive Acc (X:Type) (R:X→ X→ Prop) (x:X) : Prop :=
  Acc_intro : (∀ y : X, R y x → Acc R y) → Acc R x.
Definition well_founded := fun (X:Type) (R:X→ X→ Prop) ⇒ ∀ a:X, Acc R a.
```

Coq comes with a library for constructing well-founded relations as well as proofs that several relations on commonly used datatypes are well-founded, the $<$ relation on `nat` being the simplest example.

It is easy to construct AF relations from *decidable* well-founded relations: If we are given a decidable WF relation `R` then we will show next that the relation `fun x y ⇒ not (R y x)` is AF. This will enable us to re-use Coq libraries and lemmas for WF relations in developments for AF relations.

```
Definition dec_rel (X:Set) (R:X→ X→ Prop) := ∀ x y,{not (R y x)}+{R y x}.
```

```
Corollary af_from_wf (X:Set) (R : X → X → Prop) :
  well_founded R → dec_rel R → almost_full (fun x y ⇒ not (R y x)).
```

With this principle, and taking into account that $<$ is decidable and a total order, we can actually prove, for example:

```
Lemma leq_af : almost_full le.
```

since $\forall xy, x \leq y \leftrightarrow \neg(y < x)$ on natural numbers.

3.1 From Almost-Full to Well-Founded Relations

It is now time we saw how AF relations can be used to prove termination. The key intuition comes from online termination testing with WQOs. Recall that a WQO-based termination checker takes a WQO $\preceq$ and a "history" of past values and when presented with a new value checks whether some old value from the history is related to this new value.

Think now of the relation `T : X → X → Prop` which relates all adjacent values s_{i+1} and s_i (and only those) in the input sequence. This is often called the *transition relation* of the program that generates this sequence. As a convention we will be using the first argument of `T` as the "next" value and the second as the "current" value (so that we have $T\ s_{i+1}\ s_i$ for every i). The termination test that our WQO-based checker effectively implements is that:

$$T^+ \cap (\preceq)^{op} = \emptyset$$

where T^+ is the transitive closure of T and $(\preceq)^{op}$ is just the inverse of $\preceq$. No infinite sequence can pass this test, because an infinite sequence will necessarily have elements related by $\preceq$! Put another way, if the test succeeds the transition relation cannot have infinite chains – well, it is well-founded!

Generalizing our intuition from transition relations to arbitrary relations, and weakening the assumptions from WQOs to AF relations, the following lemma is the most important result of this paper, hence we put it in a big box:

```
Lemma wf_from_af (X:Set) (R T : X → X → Prop):
 (∀ x y, clos_trans_1n X T x y ∧ R y x → False) →
 almost_full R → well_founded T.
```

In the `wf_from_af` lemma, `clos_trans_1n X T` is just the transitive closure of `T`, as defined in Coq's standard library. Notice that it is easy to show that a relation is WF iff its transitive closure is WF, and hence the previous theorem could be restated to say that if the intersection of a transitive relation with the inverse of an AF relation is empty, then the relation is WF. Accordingly, using `wf_from_af` we can also derive the following simple lemma for *transitive* AFs (WQOs):

```
Lemma wf_from_wqo :
  ∀ (X:Set) (R : X → X → Prop), transitive X R → almost_full R →
  well_founded (fun x y ⇒ R x y ∧ not (R y x)).
```

For instance, for the $\leq$ relation on natural numbers, it is clearly the case that $\lambda xy.x \leq y \wedge \neg(y \leq x)$ is WF. This relation is simply $<$.

3.2 A New Induction Principle

If we can use lemma `wf_from_af` to form WF relations, then we can surely use it to perform induction. The theorem `af_induction` in Figure 1 demonstrates a new

```
af_induction
     : ∀ (X : Set) (T R : X → X → Prop),
       almost_full R →
       (∀  x y : X, clos_trans_1n X T x y ∧ R y x → False) →
       ∀ P : X → Set,
       (∀  x : X, (∀  y : X, T y x → P y) → P x) → ∀ a : X, P a

well_founded_induction
     : ∀ (X : Set) (T : X → X → Prop),
       well_founded T →
       ∀ P : X → Set,
       (∀  x : X, (∀  y : X, T y x → P y) → P x) → ∀ a : X, P a
```

Fig. 1. AF vs WF induction principles

induction principle, based on `wf_from_af`. Intuitively `T` is the relation between the argument in the "next" recursive call (y), and the previous (x) and we are simply requiring that the transitive closure of `T` has an empty intersection with (the inverse of) some AF relation `R`.

Hence, when using AF induction, the programmer must (i) provide an AF relation, (ii) show the emptyness of the intersection, and (iii) provide a functional. It is worth contrasting `af_induction` with well-founded induction in Figure 1. Notably, `well_founded_induction` only requires a proof that `T` is WF. As a final remark we have also developed mutual induction variations of `af_induction`, which we will not describe in this paper for lack of space.

4 Constructions on AF Relations

So far we have derived a new AF-based induction principle, and now we move on to describing some of the benefits of using AF relations for proving programs terminating. The nicest feature of AF relations is their *composability*. Together with our results from Section 3, which can be used to give "ground" AF from existing WF relations, this section presents a powerful toolkit for the `af_induction` user. As a remark, there exist similar results for classical WQOs [34,6].

4.1 AF Unions

If we are given an infinite sequence in which there exist two related elements by relation `R` then clearly these two elements are also related by `R` ∪ `T`. Thus:

```
Corollary af_union (X:Set) (R T : X → X → Prop):
 almost_full R → almost_full (fun x y ⇒ R x y ∨ T x y).
```

4.2 AF Intersections

Intersections are much more interesting. Imagine that we have AF relations `R` and `T`. If we are presented with an infinite sequence then we definitely know that R relates some elements in the sequence, and T relates some elements in the sequence, but are there any elements that are *simultaneously* related by R and T? Remarkably, the answer is affirmative. A generalization of this theorem to k-ary AF relations is often called the "intuitionistic version of Ramsey's theorem" [35].

Here we focus on the binary case. We will not present the proof in detail here; the theorem follows from a much more general theorem in Section 7.

```
Corollary af_intersection (X:Set) (R T :X→ X→ Prop):
  almost_full R → almost_full T → almost_full (fun x y ⇒ R x y ∧ T x y).
```

The binary version of the Ramsey theorem is, using classical logic, a direct consequence of `af_intersection`: consider a binary relation R on `nat` and call a subset A of `nat` homogeneous iff:

– For all n and m in A such that $n < m$ it is the case that $R\,n\,m$, *or*
– For all n and m in A such that $n < m$ it is the case that $\neg(R\,n\,m)$.

Ramsey's theorem states that for every binary relation R there exists an *infinite* homogeneous subset of `nat`, A. To prove this, assume by contradiction that no such infinite homogeneous subset exists. This means that both R and $\neg R$ are AF, which means that their intersection is AF by `af_intersection`. But the empty relation cannot be AF because it relates no elements whatsoever!

As a final remark, the classical proof of the intersection theorem for the case of WQOs is simpler due to the transitivity assumption [34,25].

4.3 Type-Based Combinators

In this section we show how to derive AF relations from simpler ones in a type-directed way, and how we may use them to define recursive functions.

Ranking Functions. We can show a theorem that is useful when we would like to map complicated data structures to `nat` values through "ranking functions".

```
Corollary af_cofmap (X Y:Set) (f:Y→ X) (R:X→ X→ Prop):
  almost_full R → almost_full (fun x y ⇒ R (f x) (f y)).
```

For instance we may map our data structures to natural numbers and re-use the $\leq$ relation and the `leq_af` witness that $\leq$ is AF.

Example 1 (Use of a ranking function). Consider the following definition (in Haskell notation here, AFExamples.v gives the Coq version):

```
flip1 (0,_) = 1
flip1 (_,0) = 1
flip1 (x+1,y+1) = flip1 (y+1,x)
```

Through the use of `af_cofmap` we may define `flip` by observing that the transition relation is `T x y := fst x <= snd y ∧ snd x < fst y`. We may now take `R x y := fst x + snd x <= fst y + snd y` as our AF relation. Showing that `∀ x y, clos_trans T x y ∧ R y x → False` is easy and the proof that R is AF is just (`af_cofmap leq_af`).

Finite Types. There is a very natural AF relation on types that have finitely many inhabitants, and that is simply the equality on elements of these types. The simplest interesting such finite type is `bool`. Why is equality on booleans AF? Because in any infinite sequence we are guaranteed that in the first three elements of the sequence two of them will be equal. Hence we can show:

```
Lemma af_bool : almost_full (@eq bool).
```

and the proof involves three applications of `AF_SUP` followed by a `AF_ZT`.

We are not going to generalize here this construction to arbitrary finite types, but the reader should be convinced that this is possible to do – a proof object with $k+1$ uses of `AF_SUP` before returning `AF_ZT` does the job for any finite type inhabited by k values. Our Coq development includes this construction.

Products. The intersection property and cofunctoriality are already extremely powerful – here is the simplest construction to create an AF relation for products based on these components:

```
Lemma af_product (X : Set) (Y : Set) :
  ∀ (R : X → X → Prop) (T : Y → Y → Prop),
  almost_full R → almost_full T →
  almost_full (fun x y ⇒ R (fst x) (fst y) ∧ T (snd x) (snd y)).
```

The proof is just applications of `af_intersection` and `af_cofmap` through the `fst` and `snd` projections out of pairs.

Of course, this is not the only AF relation on products – it's just a particular one. For instance one could completely ignore the second component of a pair and only use an AF relation on the first component though the use of `af_cofmap`.

Example 2 (Lexicographic order). Consider the definition (in Haskell notation):

```
flex (0,_) = 1
flex (_,0) = 1
flex (x+1,y+1) = f (x,y+2) + f (x+1,y)
```

This is an example of a definition where the arguments descend lexicographically. We can also observe that in any recursive call, one of the two arguments is decreasing. This immediately suggests that we should use the AF relation

```
R x y := fst x <= fst y ∧ snd x <= snd y
```

Recall that $\leq$ is AF and hence, using `af_product`, the relation `R` is AF. The transition relation of the program is also what you'd expect:

```
T x y := fst x < fst y ∨ (fst x = fst y ∧ snd x < snd y)
```

since in the first recursive call, the first argument becomes smaller, and in the second recursive call the second argument becomes smaller, while the first remains the same. It is then simple to show the proof obligations of `af_induction`.

Sums. If we are given two AF relations on types `X` and `Y` respectively, is there a natural AF relation that we can define on `X+Y`? One that we find often useful is the relation that lifts these two relations in the following way:

```
Definition sum_lift (X Y:Set) (R:X→ X→ Prop) (T:Y→ Y→ Prop) (x y:X+Y):=
  match (x,y) with
  | (inl x0, inl y0) ⇒ R x0 y0
  | (inl x0, inr y0) ⇒ False
  | (inr x0, inl y0) ⇒ False
  | (inr x0, inr y0) ⇒ T x0 y0
  end.
```

If two elements have the same tags they are compared with one or the other relation, otherwise they are not related. We have proved that if `R` and `T` are AF, then so is `sum_lift R T`. The key intuition behind our construction is the connection between tagged sums and products where the first component is the "tag" and the second is the value, and is omitted for lack of space. Our result is:

```
Corollary af_sum_lift (X Y:Set) (R:X→ X→ Prop) (T:Y→ Y→ Prop):
  almost_full R → almost_full T → almost_full (sum_lift R T).
```

We do not give here an example of `af_sum_lift`, but our Coq development includes examples that use it in the context of mutual induction.

Dependent Products and Recursive Types. We do not currently include combinators for dependent products nor recursive types, though nothing seems to be prohibitive about either. We leave this as future work, following past work on homeomorphic embeddings in classical and intuitionistic settings [25,20,3,29].

5 Size-Change Termination and AF Induction

We have examined combinators on AF relations, and simple examples such as lexicographic descent. Lexicographic orders are not terribly difficult (In fact, Coq already comes with combinators to compose lexicographically two well-founded relations) but the power of the method shows itself in examples that go beyond lexicographic orders. Consider the following example.

Example 3 (Beyond lexicographic order). Consider:

```
gnlex (0,_) = 1
gnlex (_,0) = 1
gnlex (x+1,y+1) = gnlex (y+1,y) + gnlex (y+1,x)
```

To define `gnlex` as a function, we will use the AF `R` for products, and the "obvious" transition relation `T`:

```
T x y := (fst x = snd y ∧ snd x < snd y)∨ (fst x = snd y ∧ snd x < fst y)
R x y := fst x <= fst y ∧ snd x <= snd y
```

It's now possible to show that the transitive closure of `T` has an empty intersection with the inverse of `R` and our development derives `gnlex` using `af_induction`.

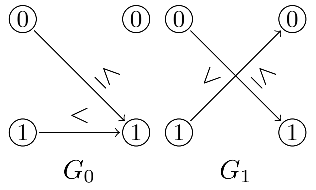
Ben-Amram [2] notices that examples like `gnlex` belong in a syntactic class of programs that can be shown terminating by size-change termination (SCT) [21,16,2] but not by a direct lexicographic descent argument, although semantically the class of *mathematical functions* one may define using SCT and those that can be defined with lexicographic descent orders coincide. It is then reassuring to see that examples from that syntactic class can be written quite straightforwardly!

5.1 Formal Connection

In fact, the connection to size-change termination can be made more precise. The short summary of this section is that the soundness of size-change termination follows from our general `wf_from_af` lemma. For the rest of this section we show this connection, using `gnlex` as our working example. This section is not formalized in our development.

The first step in showing that a recursive definition is terminating with SCT is to identify the various recursion patterns and abstract each as a *size-change graph.* A size-change graph for a k-argument function is a labeled graph with nodes labeled from $\{0, \ldots, k-1\}$ and arcs with labels $<$ and $\leq$.

Example 4 (Size-change graph for gnlex). For our two-argument `gnlex` we get the following two size-change graphs:



A size-change graph G for a k-argument function schema induces a relation on k-tuples and we say that a size-change graph *approximates* a relation `T` iff $T \subseteq T_G$. In our `gnlex` example, each of the two graphs approximates a disjunct from `T`.

Size change graphs *compose* so that the composition of two arcs one of which is $<$ creates a new arc $<$, whereas the composition of two $\leq$ arcs gives a new $\leq$ arc. We write this composition with notation $G_1; G_2$ (written G_{12} for brevity below). Graph composition satisfies the following proposition.

Proposition 1. *If G_1 approximates T_1 and G_2 approximates T_2 then $G_1; G_2$ approximates $T_1 \cdot T_2$ (where $\cdot$ is transitive relation composition).*

Assume now that the transition relation of a program is given by n-disjuncts $T = T_1 \cup \ldots T_n$ each of which corresponds to some recursion pattern and is approximated by a size-change graph G_i (as in our example with $n = 2$). Size-change termination then considers the set S, defined as the transitive closure of the set $\{G_1, \ldots, G_n\}$ under graph composition.

Example 5 (Transitive closure of size-change graphs). What is this set S in our `gnlex` example? If we start off with G_0 and G_1, we have to consider the compositions $G_0; G_0$, $G_0; G_1$, $G_1; G_0$, and $G_1; G_1$. We observe that G_{00} is a new graph with edges $0 \xrightarrow{\leq} 1$, $1 \xrightarrow{<} 1$, G_{01} is a new graph with edges $0 \xrightarrow{<} 0$, $1 \xrightarrow{<} 0$, G_{10} is *exactly* G_{00} and G_{11} is a new graph with edges $0 \xrightarrow{<} 0$, $1 \xrightarrow{<} 1$. If we continue in this fashion we can compute that the set S is just:

$$S = \{G_0, G_1, G_{00}, G_{01}, G_{11}, G_{111}\}$$

What is the importance of the set S? We have seen that T can be approximated by $\{G_0, G_1\}$ and we have seen that compositions of graphs approximate compositions of relations. This means that for every k, the composition of T with itself k times T^k (which we will call the k-th *power* of T) can be approximated by the set of graphs in S (which will typically, as in our example, be finite): Precisely, for every x and y such that $T^k\ x\ y$ it is the case that $T_G\ x\ y$ for some $G \in S$. This, in turn, enables the following lemma.

Lemma 1. *Assume that $T = T_1 \cup \ldots \cup T_n$ and G_i approximates T_i, and let S be the transitive closure of the set $\{G_i, \ldots, G_n\}$. If every $G \in S$ induces a relation T_G such that $T_G \cap R^{op} = \emptyset$ for some AF R then T is well-founded.*

Proof. By `wf_from_af` we only have to show that for all x and y such that $T^+ x\ y$ it is not the case that $R\ y\ x$. If $T^+ x\ y$ then there exists some k such that $T^k x\ y$, hence there exists some $G \in S$ such that $T_G\ x\ y$ and we know that $T_G \cap R^{op} = \emptyset$.

Next, consider the size-change graph I with edges $i \xrightarrow{\leq} i$ for each i, and let us call the induced relation $T_I\ x\ y = \bigwedge x_i \leq y_i$. By `af_intersection`, T_I is AF.

Example 6. We can now show that `gnlex` is terminating by checking that every graph $G \in S$ has empty intersection with $(T_I)^{op}$ and using Lemma 1.

Size-change termination uses the same AF relation T_I and Lemma 1, through the following auxiliary lemma.

Lemma 2. *If G approximates T and some power G^n of G contains an arc $i \xrightarrow{<} i$ then $T \cap T_I^{op} = \emptyset$.*

Proof. Assume that $T\ x\ y$ and $T_I\ y\ x$. We then have $(T{\cdot}T_I)\ x\ x$ and $(T{\cdot}T_I)^n\ x\ x$. But I approximates T_I and because compositions of graphs approximate compositions of relations and $G; I = G$ it follows that G^n approximates $(T{\cdot}T_I)^n$. this means that $x_i < x_i$, which is a contradiction.

Theorem 1 (Size-change termination). *Assume that $T = T_1 \cup \ldots \cup T_n$ and G_i approximates T_i, and let S be the transitive closure of the set $\{G_i, \ldots, G_n\}$. If every $G \in S$ has a power with an arc $i \xrightarrow{<} i$ then T is well-founded.*

Proof. By Lemma 2 we know that $T_G \cap T_I^{op} = \emptyset$ for every $G \in S$, and by Lemma 1 we are done.

Hence, we have proved the SCT condition using the `wf_from_af` theorem. The reader can observe that the condition is true for the set S we have computed for `gnlex`. Finally, the SCT criterion is often stated by requiring that every idempotent graph $G \in S$ has an arc $i \xrightarrow{<} i$, which is an equivalent condition, since any size-change graph has an idempotent power.

5.2 Mutually Recursive Definitions

What about mutual induction schemes, a common application of SCT? The modifications to our previous setup are small: For mutual induction schemes each of the size-change graphs can be extended so that each argument tuple is paired up with a *tag*, drawn from a finite set of tags, each corresponding to a definition in a group of mutual definitions. Composing transitively a graph with another is possible when the target of the first and source of the second graphs are also equal. The SCT criterion then requires that every graph G in the transitive closure of the size-change graphs with *the same* source and target tags has a power with an arc $i \xrightarrow{\leq} i$. The proof is an extension of our previous proof using the AF relation $T_I\ (f,x)\ (g,y) = (\bigwedge x_i \leq y_i) \wedge f = g$, where f and g are the tags. This relation is AF because of `af_intersection` and the fact that equality on finite types (the finite set of tags) is AF.

6 The Terminator Rule

We have used online termination and WQOs as a way to approach AF relations and `af_induction`, but it turns out that `af_induction` is general enough to capture the proof principle behind Terminator [9,26]. The key theorem behind Terminator is the following *disjunctive well-foundedness* proposition: If $R^+ \subseteq R_1 \cup \ldots \cup R_n$ and $R_1 \ldots R_n$ are well-founded then so is R .

The proof relies on a Ramsey argument [26], but here we have proved it – intuitionistically – from theorem `wf_from_af` from Section 3.1. In the case where $n = 2$ it suffices to instantiate `wf_from_af` with

```
R x y := not (R1 y x) ∧ not (R2 y x)
```

We can easily then use disjunctive well-foundedness to deduce the standard Terminator proof rule (for the union of two WF relations):

```
Lemma disj_wf_induction:
 ∀ (X:Set) (T : X → X → Prop)
 (R1 R2 : X → X → Prop) (decR1:dec_rel R1) (decR2:dec_rel R2),
 well_founded R1 → well_founded R2 →
 (∀  x y, clos_trans_1n X T x y → R1 x y ∨ R2 x y) →
 ∀ P : X → Set, (∀  x, (∀  y, T y x → P y) → P x) → ∀ a, P a.
```

7 Generalized Ramsey's Theorem

We now turn our attention to the intersection theorem for AF relations, given in Section 4.2. We will show here a much more general result for relations of inductive arities. We start with defining predicates on lists

```
Definition LRel (X:Set) := list X → Prop.
```

We can generalize our `almost_full` definition for predicates over lists:

```
Inductive almost_full_l X : LRel X → Prop :=
 | AF_ZT : ∀ (R : LRel X), (∀ xs, R xs) → almost_full_l R
 | AF_SUP : ∀ (R : LRel X),
   (∀ x, almost_full_l (fun ys ⇒ R ys ∨ R (x::ys))) → almost_full_l R.
```

Again there exist two constructors – the `AF_ZT` constructor asserts that the predicate is true for every list, whereas the `AF_SUP` constructor asserts that when presented with one new element `x`, the predicate `fun ys ⇒ R ys ∨ R (x :: ys)` is AF. We will be interested in predicates over lists which have an "inductive arity", which we formalize with the definitions below:

```
Inductive WFT (X:Set):=
  | ZT: WFT X
  | SUP: (X→ WFT X)→ WFT X.
```

```
Fixpoint Arity(X:Set) (p:WFT X) (R:LRel X):=
  match p with
  | ZT ⇒ ∀ ys, R ys ↔ R nil
  | SUP w ⇒ ∀ x,
    Arity (w x) (fun ys ⇒ R (x::ys))
  end.
```

`WFT` encodes well-founded trees over a set `X`. The `Arity` fixpoint defines when a relation `R` has an "inductive" arity. If we are given a `ZT` well-founded tree then the truth value of the predicate `R` is constant. However if we are given a `SUP w` tree then for every element `x`, the relation `fun ys ⇒ R (x::ys)` has an inductive arity. One can see that since `WFT` is inductive this fixpoint ensures that after a finite number of inputs (which nevertheless depends on the individual input sequence each time) the value of the predicate will become constant.

Our main result is the following:

```
Lemma af_intersection_l_cor (X:Set):
  ∀ (p : WFT X) R T, Arity p R → Arity p T →
  almost_full_l R → almost_full_l T → almost_full_l (fun xs⇒ R xs ∧ T xs).
```

Its proof is done by generalizing the statement to:

```
Lemma af_intersection_l (X:Set):
  ∀ (p:WFT X),
  ∀ (R:LRel X), almost_full_l R → ∀ (T:LRel X), almost_full_l T →
  ∀ (C:LRel X), ∀ A B, Arity p A → Arity p B →
  (∀ xs, R xs → C xs ∨ A xs) → (∀ xs, T xs → C xs ∨ B xs) →
  almost_full_l (fun xs ⇒ C xs ∨ A xs ∧ B xs).
```

and proceeding by induction on the arity witness `p`, and then the AF proof for `R`, and then the AF proof for `T`.

We can move to and from `LRel`s with inductive arities. For instance, for a binary relation `R` we may define an `LRel (BinRelExtension R)` so that the value of the predicate on any list with more than one elements is the value of the relation on the first two. The value on smaller lists is just `False`. With this definition in place we have proved the following:

```
Corollary af_l_af (X:Set) (R : X → X → Prop) :
   almost_full_l (BinRelExtension R) ↔ almost_full R.
```

from which the usual `af_intersection` theorem from Section 4.2 follows.

8 Related and Future Work

We have discused the Terminator rule [9,26] in Section 6, which is itself related to the proposition that if the union of WF is transitive, then it is also WF [14]. Why has Terminator been so successful? One answer is *composability*: The way the implementation works is by trying to iteratively synthesize WF relations $R_1 \ldots R_n$ so that $R^+ \subseteq R_1 \cup \ldots \cup R_n$, where R is the transition relation of the program, starting from $\emptyset$, and unioning-up WF relations until the proof goes through. There has also been work on (classical) proofs of SCT in the context of Isabelle [18]: the author leaves the problem of justifying SCT constructively as open – this is the problem we have solved in our work.

Porting Ramsey theory in a constructive setting seems to have been a fascinating subject among mathematicians and computer scientists, since the original proof and definitions seem hopelessly classical. Our development is based on Bezem and Veldman's original ideas [35]; however unlike their work we do not postulate bar induction and our AF relations are over arbitrary sets, not natural numbers. Side-stepping bar induction, our definition of AF relations directly encodes all possible "choices". This is similar to previous work [10,3,29,13], which also does not postulate bar induction but rather *inductively defines* bars inside type theory. Finally, AF relations have further – independent of termination – uses, for instance one may define a set to be finite inside type theory iff the equality on elements of that set is AF [11,24].

Nowadays there exists a large set of recursion-encoding techniques in type theory and Coq, some of which include good support for automation. The most straightforward way to program recursion in Coq [4] is either by structural recursion or by using subset types [32] and `measure` arguments. The Coq CoLoR library [5] can be used to manipulate well-founded relations and measures. An extension of "guarded" recursion (and co-recursion) implemented in a variant of Agda is sized-types [1] (not to be confused with size-change termination).

The Bove and Capretta method [7] is the *de-facto* way to define complex recursive programs in Type Theory: For each definition the user introduces an indexed type family with constructors corresponding to the recursive calls. After-the-fact, she can provide such an inductive witness at the call-sites. Krauss [19] proposes a related technique for showing automatically the termination of Isabelle functions by extracting their *inductive graph* and using induction on that graph. In followup work, Krauss *et al.* [17] show how to re-use termination proofs for term rewrite systems to certify the termination of Isabelle functions.

Charguéraud [8] has recently presented a well-engineered library that uses a measure-based fixpoint combinator inspired from recursion theory (on "optimal fixpoints"). Finally, Megacz [23] gave a monadic way to write recursive definitions using a coinductive type, which allows one to prove that the definition will terminate after-the-fact.

8.1 Directions for Future Work

We have already mentioned several possibilities for future work. An important direction is the design of better variations on AF mutual induction. Another ambitious direction is tool support and automation, in order to help the user synthesize a termination argument (perhaps driven by failed proofs, as Terminator), or to automatically discharge the generated relation inclusion obligations. Finally, we would like to further improve the practicality of our method; for example have automatic derivation of function simplification theorems.

Acknowledgments. We would like to thank Carsten Fuhs and the anonymous reviewers for many helpful comments. Part of this work was funded by the Swedish Research Council, project reg no: 2008:6902.

References

1. Abel, A.: Termination and productivity checking with continuous types. In: Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications, TLCA 2003, Valencia, Spain, pp. 1–15. Springer, Heidelberg (2003), `http://dl.acm.org/citation.cfm?id=1762348.1762349`, ISBN: 3-540-40332-9
2. Ben-Amram, A.M.: General Size-Change Termination and Lexicographic Descent. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 3–17. Springer, Heidelberg (2002)
3. Berghofer, S.: A Constructive Proof of Higman's Lemma in Isabelle. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 66–82. Springer, Heidelberg (2004)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
5. Blanqui, F., Koprowski, A.: Color: a COQ library on well-founded rewrite relations and its application to the automated verification of termination certificates. MSCS 21(4), 827–859 (2011)
6. Bolingbroke, M., Jones, S.P., Vytiniotis, D.: Termination combinators forever. In: Proceedings of the 4th ACM Symposium of Haskell 2011, pp. 23–34. ACM (2011)
7. Bove, A., Capretta, V.: Modelling general recursion in type theory. MSCS 15, 671–708 (2005)
8. Charguéraud, A.: The Optimal Fixed Point Combinator. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 195–210. Springer, Heidelberg (2010)
9. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proceedings of PLDI 2006, pp. 415–426. ACM (2006)
10. Coquand, T.: An analysis of Ramsey's Theorem. Inf. Comput. 110, 297–304 (1994)
11. 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)
12. Dershowitz, N., Lindenstrauss, N., Sagiv, Y., Serebrenik, A.: A general framework for automatic termination analysis of logic programs. Appl. Algebra Eng. Commun. Comput. 12(1/2), 117–156 (2001)
13. Fridlender, D.: Higman's Lemma in Type Theory. In: Giménez, E. (ed.) TYPES 1996. LNCS, vol. 1512, pp. 112–133. Springer, Heidelberg (1998)

14. Geser, A.: Relative termination. PhD thesis, Universität Passau (1990)
15. Heizmann, M., Jones, N.D., Podelski, A.: Size-Change Termination and Transition Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
16. Jones, N.D., Bohr, N.: Call-by-value termination in the untyped lambda-calculus. Logical Methods in Computer Science 4(1) (2008)
17. 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)
18. Krauss, A.: Certified Size-Change Termination. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
19. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning 44, 303–336 (2010)
20. Kruskal, J.B.: Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. Trans. Amer. Math. Soc. 95, 210–225 (1960)
21. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proceedings of POPL 2001, pp. 81–92. ACM (2001)
22. Leuschel, M.: Homeomorphic Embedding for Online Termination of Symbolic Methods. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 379–403. Springer, Heidelberg (2002)
23. Megacz, A.: A coinductive monad for Prop-bounded recursion. In: Stump, A., Xi, H. (eds.) Proceedings of PLPV 2007, pp. 11–20. ACM (2007)
24. Bezem, M.A., Nakata, K., Uustalu, T.: On streams that are finitely red. To appear in Logical Methods in Computer Science (2011)
25. Nash-Williams, C.S.J.A.: On well-quasi-ordering finite trees. In: Mathematical Proceedings of the Cambridge Philosophical Society, vol. 59, pp. 833–835. Cambridge Univ. Press (1963)
26. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proceedings of LICS 2004, pp. 32–41. IEEE Computer Society, Washington, DC (2004)
27. Richman, F., Stolzenberg, G.: Well-quasi-ordered sets. Advanced Mathematics, 145–193 (1993)
28. Sagiv, Y.: A termination test for logic programs. In: ISLP, pp. 518–532 (1991)
29. Seisenberger, M.: An Inductive Version of Nash-Williams' Minimal-Bad-Sequence Argument for Higman's Lemma. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 233–242. Springer, Heidelberg (2002)
30. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In: Proceedings of ICFP 2007, pp. 71–84. ACM (2007)
31. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Proceedings of ILPS 1995, The International Logic Programming Symposium, pp. 465–479. MIT Press (1995)
32. Sozeau, M.: Subset Coercions in COQ. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007)
33. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning 2(1), 41–62 (2009)
34. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
35. Veldman, W., Bezem, M.: Ramsey's theorem and the pigeonhole principle in intuitionistic mathematics. Journal of the London Mathematical Society s2-47(2), 193–211 (1993)

Certification of Nontermination Proofs*

Christian Sternagel[1] and René Thiemann[2]

[1] Japan Advanced Institute of Science and Technology, Japan
[2] Institute of Computer Science, University of Innsbruck, Austria

Abstract. Automatic tools for proving (non)termination of term rewrite systems, if successful, deliver proofs as justification. In this work, we focus on how to certify nontermination proofs. Besides some techniques that allow to reduce the number of rules, the main way of showing nontermination is to find a loop, a finite derivation of a special shape that implies nontermination. For standard termination, certifying loops is easy. However, it is not at all trivial to certify whether a given loop also implies innermost nontermination. To this end, a complex decision procedure has been developed in [1]. We formalized this decision procedure in Isabelle/HOL and were able to simplify some parts considerably. Furthermore, from our formalized proofs it is easy to obtain a low complexity bound. Along the way of presenting our formalization, we report on generally applicable ideas that allow to reduce the formalization effort and improve the efficiency of our certifier.

Keywords: nontermination, formalization, interactive theorem proving, term rewriting.

1 Introduction

In program verification the focus is on proving that a function satisfies some property, e.g., termination. However, in presence of a *bug* it is more important to find a counterexample indicating the problem. In this way, we can save a lot of time by abandoning a verification attempt as soon as a counterexample is found. In term rewriting, a well known counterexample for termination is a loop, essentially giving some "input" on which a "program" does not terminate. As soon as specific evaluation strategies are considered it might not be easy to verify whether a given loop constitutes a proper counterexample. However, since many programming languages employ an eager evaluation strategy, methods for proving innermost nontermination are important. What is more, some very natural functions are not even expressible without evaluation strategy. Take for example equality on terms. There is no (finite) term rewrite system (TRS) that encodes equality on arbitrary terms (the problem is the case where the two given terms are different). Using innermost rewriting, encoding equality is possible by the following four rules, as shown by Daron Vroon (personal communication; he used this encoding to properly model the built-in equality of ACL2).

* This research is supported by the Austrian Science Fund (FWF): J3202, P22767.

$$x == y \to \mathsf{chk}(\mathsf{eq}(x,y)) \quad (1) \qquad \mathsf{chk}(\mathsf{true}) \to \mathsf{true} \quad (3)$$
$$\mathsf{eq}(x,x) \to \mathsf{true} \quad (2) \qquad \mathsf{chk}(\mathsf{eq}(x,y)) \to \mathsf{false} \quad (4)$$

Current techniques for proving innermost nontermination of TRSs consist of preprocessing techniques (narrowing the search space by removing rules) followed by finding a loop, for which the complex decision procedure of [1] allows to decide whether it implies innermost nontermination. We formalized this decision procedure as part of our ***Isa**belle **F**ormalization **o**f **R**ewriting* (IsaFoR). The corresponding certifier CeTA can be obtained by Isabelle/HOL's code generator [2,3]. Both IsaFoR and CeTA are freely available at `http://cl-informatik.uibk.ac.at/software/ceta/` (the relevant theories for this paper are `Innermost_Loops` and `Nontermination`, together with their respective implementation theories, indicated by the suffix `_Impl`).

During our formalization we were able to simplify some parts of the decision procedure considerably. Mostly, due to a new proof which, in contrast to the original proof, does not depend on Kruskal's tree theorem. As a result, we can replace the most complicated algorithm of [1] by a single line. Moreover, we report on how we managed to obtain efficient versions of other algorithms from [1] within Isabelle/HOL [4].

The remainder is structured as follows. In Sect. 2 we give preliminaries. Then, in Sect. 3, we describe the preprocessing techniques (narrowing the search space for finding a loop) that are supported by our certifier. Afterwards, we present details on loops w.r.t. the innermost strategy in Sect. 4. The main part of this paper is on our formalization of the decision procedure for innermost loops in Sect. 5, before we conclude in Sect. 6.

2 Preliminaries

We assume basic familiarity with term rewriting [5]. Nevertheless, we shortly recapitulate what is used later on. A *term* t (ℓ, r, s, u, v) is either a *variable* x (y, z) from the set $\mathcal{V}$, or a *function symbol* f (g) from the disjoint set $\mathcal{F}$ applied to some argument terms $f(t_1,\ldots,t_n)$. The *root* of a term is defined by $\mathit{root}(x) = x$ and $\mathit{root}(f(t_1,\ldots,t_n)) = f$. The set $\mathit{args}(t)$ of *arguments* of t is defined by the equations $\mathit{args}(x) = \emptyset$ and $\mathit{args}(f(t_1,\ldots,t_n)) = \{t_1,\ldots,t_n\}$. The set of *variables occurring in a term* t is denoted by $\mathcal{V}(t)$. A *context* C (D) is a term containing exactly one occurrence of the special *hole* symbol $\Box$. Replacing the hole in a context C by a term t is written $C[t]$. The term t is a *(proper) subterm* of the term s, written ($s \rhd t$) $s \unrhd t$, iff there is a (non-hole) context C such that $s = C[t]$, iff there is a (non-empty) position p such that $s|_p = t$. We write $s \unrhd_{\mathcal{F}} t$ iff $s \unrhd t$ and $t \notin \mathcal{V}$. A *substitution* σ (μ) is a mapping from variables to terms whose *domain* $\mathit{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. The *range* of a substitution is $\mathit{ran}(\sigma) = \{\sigma(x) \mid x \in \mathit{dom}(\sigma)\}$. We represent concrete substitutions using the notation $\{x_1/t_1,\ldots,x_n/t_n\}$. We use σ interchangeably with its homomorphic extension to terms, writing, e.g., $t\sigma$ to denote the application of the substitution σ to the term t. A *(rewrite) rule* is a pair of terms $\ell \to r$

and a term rewrite system (TRS) $\mathcal{R}$ is a set of such rules. The *rewrite relation (induced by $\mathcal{R}$)* $\rightarrow_{\mathcal{R}}$ is defined by $s \rightarrow_{\mathcal{R}} t$ iff there is a context C, a rewrite rule $\ell \rightarrow r \in \mathcal{R}$, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. Here, we call $\ell\sigma$ a *redex* (short for *reducible expression*) and sometimes write $s \rightarrow_{\mathcal{R},\ell\sigma} t$ to make it explicit. A *normal form* is a term that does not contain any redexes. When a rewrite step $s \rightarrow_{\mathcal{R},\ell\sigma} t$ additionally satisfies that all arguments of $\ell\sigma$ are normal forms, it is called an *innermost (rewrite) step*, written $s \overset{\mathsf{i}}{\rightarrow}_{\mathcal{R}} t$. We freely drop $\mathcal{R}$ from $s \rightarrow_{\mathcal{R}} t$ if it is clear from the context.

A term t is (innermost) nonterminating w.r.t. $\mathcal{R}$, iff there is an infinite (innermost) rewrite sequence starting at t, i.e., a derivation of the form

$$t = t_1 \overset{(\mathsf{i})}{\rightarrow}_{\mathcal{R}} t_2 \overset{(\mathsf{i})}{\rightarrow}_{\mathcal{R}} t_3 \overset{(\mathsf{i})}{\rightarrow}_{\mathcal{R}} \cdots$$

A TRS $\mathcal{R}$ is (innermost) nonterminating iff there is a term t that is (innermost) nonterminating w.r.t. $\mathcal{R}$.

3 A Framework for Certifying Nontermination

As for termination, there are several techniques that may be combined in order to prove nontermination. On the one hand, there are basic techniques, i.e., those that immediately prove nontermination; and on the other hand, there are transformations, i.e., mappings that turn a given TRS $\mathcal{R}$ into a transformed TRS $\mathcal{R}'$ (for which, proving nontermination is hopefully easier). Such transformations are *complete* iff (innermost) nontermination of $\mathcal{R}'$ implies (innermost) nontermination of $\mathcal{R}$. In order to prove nontermination, arbitrary complete transformations can be applied, before finishing the proof by a basic technique.

In our development we formalized the following basic techniques and complete transformations. Except for innermost loops and string reversal, none of these techniques posed any difficulties in the formalization.

Well-Formedness Check. A TRS $\mathcal{R}$ is *(weakly) well-formed* iff no left-hand side is a variable and all (applicable) rules $\ell \rightarrow r$ satisfy $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$. Where a rule is *applicable* iff the arguments of its left-hand side are normal forms (otherwise the rule could never be used in the innermost case).

Lemma 1. *If $\mathcal{R}$ is not (weakly) well-formed, it is (innermost) nonterminating.*

Thus a basic technique is to check whether a TRS is (weakly) well-formed and conclude (innermost) nontermination, if it is not.

Finding Loops. The second basic technique is to find a loop and it is treated in more detail in Sect. 4.

Rule Removal. One way to narrow the search space when trying to prove nontermination, is to get rid of rules that cannot contribute to any infinite derivation. This can be done by employing the same techniques that are already known from termination, namely monotone reduction pairs [6,7].

String Reversal. A special variant of TRSs are *string rewrite systems*, where all function symbols are fixed to be unary. For this special case, *string reversal* (see, e.g., [8] and [9] for its formalization) can be applied.

Dependency Pair Transformation. As for termination, also for nontermination, it is possible to switch from TRSs to *dependency pair problems* (DPPs) [10]. This is done by the so called *dependency pair transformation*, which intuitively, identifies the mutually recursive dependencies of rewrite rules and makes them explicit in a second set of rewrite rules, the *dependency pairs.*

For nontermination of (innermost) DPPs, we support the following techniques:

Finding Loops. For DPPs $(\mathcal{P}, \mathcal{R})$ the search space for finding loops is further restricted by the fact that pairs from $\mathcal{P}$ are only applied at the root position.

Rule Removal. Also for DPPs it is possible to narrow the search space by employing reduction pairs to remove pairs and rules that do not contribute to any infinite derivation. Note that for nontermination analysis, also the dependency graph processor and the usable rules processor do just remove pairs and rules.

Note. Since $\mathcal{R}$ is (innermost) nonterminating (by the well-formedness check) whenever $\mathcal{R}$ contains a rule $x \to r$ for some $x \in \mathcal{V}$, we only consider TRSs where all left-hand sides of rules are not variables in the remainder.

4 Loops

Loops are derivations of the shape $t \to_{\mathcal{R}}^{+} C[t\mu]$. They always imply nontermination where the corresponding infinite reduction is

$$t \to_{\mathcal{R}}^{+} C[t\mu] \to_{\mathcal{R}}^{+} C[C\mu[t\mu^2]] \to_{\mathcal{R}}^{+} C[C\mu[C\mu^2[t\mu^3]]] \to_{\mathcal{R}}^{+} \cdots \tag{5}$$

A TRS which admits a loop is called *looping.*

Note that for innermost rewriting, loopingness does not necessarily imply nontermination, since the innermost rewrite relation is not closed under substitutions. More precisely, it is not enough to have an “innermost loop” of the form $t \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{+} C[t\mu]$, since this does not necessarily imply an infinite sequence (5) when restricting to innermost rewriting. Therefore, in [1], the notion of an *innermost loop* was introduced. To facilitate the certification of innermost loops (i.e., to decide for a given loop, whether it is innermost or not), we need its constituting steps, i.e., a derivation of length $m > 0$ with redexes $\ell_i\sigma_i$:

$$t = t_1 \to_{\mathcal{R},\ell_1\sigma_1} t_2 \to_{\mathcal{R},\ell_2\sigma_2} \cdots \to_{\mathcal{R},\ell_m\sigma_m} t_{m+1} = C[t\mu] \tag{6}$$

Definition 2 (Innermost Loops). *A loop (6) is an* innermost loop *iff for all $1 \leqslant i \leqslant m$ and $n \in \mathbb{N}$, the term $\ell_i\sigma_i\mu^n$ is an innermost redex.*

That is, no matter how often μ is applied, all steps should be innermost.

Lemma 3. *A loop (6) is an innermost loop iff (5) is an innermost derivation.*

Corollary 4. *An innermost loop implies innermost nontermination.*

Note that for every loop (6) and all $n \in \mathbb{N}$, the term $\ell_i\sigma_i\mu^n$ is a redex. Hence, to make sure that those redexes are innermost, it suffices to check whether all arguments of $\ell_i\sigma_i\mu^n$ are normal forms for all $n \in \mathbb{N}$. Since $\ell_i\sigma_i$ is not a variable (we ruled out variables as left-hand sides of $\mathcal{R}$) this is equivalent to checking that for all arguments t of $\ell_i\sigma_i$, the term $t\mu^n$ is a normal form for all $n \in \mathbb{N}$. Thus, to decide whether a loop is innermost, we can use the following characterization.

Lemma 5. *Let $\mathcal{R}$ be a TRS, (6) a loop, and $\mathcal{A} = \bigcup_{1 \leqslant i \leqslant m} args(\ell_i\sigma_i)$ the set of arguments of redexes in (6). Then, (6) is an innermost loop, iff for all $t \in \mathcal{A}$ and $n \in \mathbb{N}$ the term $t\mu^n$ is a normal form, iff for all $t \in \mathcal{A}$ and $\ell \to r \in \mathcal{R}$ the term $t\mu^n$ does not contain a redex $\ell\sigma$ for any $n \in \mathbb{N}$ and σ.*

Hence, we can easily check, whether a loop is innermost, whenever for two terms t and ℓ, and a substitution μ, we can solve the problem whether there exist n and σ, such that $t\mu^n$ contains a redex $\ell\sigma$. Such problems are called *redex problems* and a large part of [1] is devoted to develop a corresponding decision procedure.

Example 6. Consider a loop $t \to^+ C[t\mu]$ for a TRS $\mathcal{R}$ containing rules (1)-(4), where $\mu = \{x/\mathsf{cons}(z,y), y/\mathsf{cons}(z,x), z/\mathsf{0}\}$. Let $D[\mathsf{chk}(\mathsf{eq}(x,y))] \to D[\mathsf{false}]$ be a step of the loop. Then, for an innermost loop we must ensure that the term $\mathsf{eq}(x,y)\mu^n$ does not contain a redex w.r.t. $\mathcal{R}$, especially not w.r.t. rule (2).

The above decision procedure works in three phases: first, redex problems are simplified into a set of matching problems. Then, a modified matching algorithm is employed, where in the end identity problems have to be solved. Finally, a decision procedure for identity problems is applied.

In the remainder, let μ be an arbitrary but fixed substitution (usually originating from some loop $t \to_{\mathcal{R}}^+ C[t\mu]$).

Definition 7 (Redex, Matching, and Identity Problems). *Let s, t, and ℓ be terms. Then a* redex problem *is a pair $t \mathrel{|\!\!>} \ell$, a* generalized matching problem *is a set of pairs $\{t_1 \gtrdot \ell_1, \ldots, t_k \gtrdot \ell_k\}$ (we call a generalized matching problem having only one pair, a* matching problem*, and drop the surrounding braces), and an* identity problem *is a pair $s \approxeq t$.*

A redex problem $t \mathrel{|\!\!>} \ell$ is solvable *iff there is a context C, a substitution σ, and an $n \in \mathbb{N}$ such that $t\mu^n = C[\ell\sigma]$. A (generalized) matching problem is* solvable *iff there is a substitution σ and an $n \in \mathbb{N}$ such that $t_i\mu^n = \ell_i\sigma$ for all pairs $t_i \gtrdot \ell_i$. An identity problem is* solvable *iff there is an $n \in \mathbb{N}$ such that $s\mu^n = t\mu^n$. In those respective cases, we call (C, σ, n), (σ, n), and n, the* solution.

5 Formalization

In [11] a straightforward certification algorithm for loops is described which does nothing else than checking rewrite steps. We extend this result significantly by also formalizing the necessary machinery to decide whether a loop is innermost. In the following, we discuss the three phases of the decision procedure from [1].

From Redex Problems to Matching Problems. A redex problem $t \mathrel{|\!\!>} \ell$ with $\ell \in \mathcal{V}$ is trivially solvable using the solution $(\Box, \{\ell/t\}, 0)$. Thus, in the following we assume that $\ell \notin \mathcal{V}$. Then, solvability of $t \mathrel{|\!\!>} \ell$ is equivalent to the existence of a non-variable subterm s of $t\mu^n$ such that $s = \ell\sigma$ (i.e., ℓ matches s). In order to simplify redex problems, we represent these subterms in a finite way and consequently generate only finitely many matching problems.

Either, s starts inside t, so $s = u\mu^n$ for some $u \trianglelefteq_{\mathcal{F}} t$, or s is completely inside μ^n. But then, it must be of the form $u\mu^n$ for some $u \trianglelefteq_{\mathcal{F}} x\mu$ and x in $\mathcal{W}(t) = \bigcup_n \mathcal{V}(t\mu^n)$, where $\mathcal{W}(t)$ collects all variables which can possibly occur in a term of the form $t\mu^n$. In both cases, the equality $s = \ell\sigma$ can be reformulated to $u\mu^n = \ell\sigma$, i.e., solvability of the matching problem $u \gtrdot \ell$. In total, the redex problem is solvable iff one of the matching problems $u \gtrdot \ell$ is solvable for some $u \in \mathcal{U}(t)$, where $\mathcal{U}(t) = \{u \mid t \trianglerighteq_{\mathcal{F}} u \text{ or } x\mu \trianglerighteq_{\mathcal{F}} u \wedge x \in \mathcal{W}(t)\}$.

The following theorem (whose formalization was straightforward), corresponds to [1, Theorem 10].

Theorem 8. *Let $t \mathrel{|\!\!>} \ell$ be a redex problem. Let*

$$\mathcal{M}_{init}(t, \ell) = \textsf{if } \ell \in \mathcal{V} \textsf{ then } \{t \gtrdot \ell\} \textsf{ else } \{u \gtrdot \ell \mid u \in \mathcal{U}(t)\}$$

be the set of initial matching problems. Then $t \mathrel{|\!\!>} \ell$ is solvable iff one of the matching problems in $\mathcal{M}_{init}(t, \ell)$ is solvable.

Example 9. Continuing with Example 6, from each redex problem $\mathsf{eq}(x, y) \mathrel{|\!\!>} \ell$ we obtain the matching problems $\mathsf{eq}(x, y) \gtrdot \ell$, $\mathsf{cons}(z, y) \gtrdot \ell$, $\mathsf{cons}(z, x) \gtrdot \ell$, and $\mathsf{0} \gtrdot \ell$ where ℓ is an arbitrary left-hand side of the TRS.

Theorem 8 shows a way to convert redex problems into matching problems. However, for certification, it remains to develop an algorithm that actually computes $\mathcal{M}_{init}$. To this end, we need to compute $\mathcal{U}(t)$, which in turn requires to enumerate all subterms of a term and to compute $\mathcal{W}(t)$. Whereas the former is straightforward, computing $\mathcal{W}(t)$ is a bit more difficult: its original definition contains an infinite union.

Note that $\mathcal{W}(t)$ is only finite since we restrict to substitutions of finite domain and can be computed by a fixpoint algorithm: iteratively compute $\mathcal{V}(t)$, $\mathcal{V}(t\mu)$, $\mathcal{V}(t\mu^2), \ldots,$ until some $\mathcal{V}(t\mu^k)$ is reached where no new variables are detected. In principle, it is possible to formalize this algorithm directly, but we expect such a formalization to require tedious manual termination and soundness proofs. Thus instead, we characterize $\mathcal{W}(t)$ by the following reflexive transitive closure.

Lemma 10. *Let $R = \{(x, y) \mid x \neq y, x \in \mathcal{V}, y \in \mathcal{V}(x\mu)\}$. Then $\mathcal{W}(t) = \{y \mid \exists x \in \mathcal{V}(t), (x, y) \in R^*\}$.*

Note that R in Lemma 10 can easily be computed since whenever $(x, y) \in R$ then $x \in dom(\mu)$. Moreover, R is finite since we only consider substitutions of finite domain. Hence, the above characterization allows us to compute $\mathcal{W}$ by the algorithm of [12] (generating the reflexive transitive closure of finite relations).

Note that $\mathcal{W}(t)$ can also be defined inductively as the least set such that $\mathcal{V}(t) \subseteq \mathcal{W}(t)$ and $x \in \mathcal{W}(t) \implies \mathcal{V}(x\mu) \subseteq \mathcal{W}(t)$. And whenever a finite set

S is defined inductively, instead of implementing an executable algorithm for S manually, it might be easier to characterize S via reflexive transitive closures and afterwards execute it via the algorithm of [12]. This approach is not restricted to $\mathcal{W}$: it has been applied in the next paragraph and also in other parts of IsaFoR.

An alternative might be Isabelle/HOL's predicate compiler [13]. It can be used to obtain executable functions for inductively defined predicates and sets. However, without manual tuning we were not able to obtain appropriate equations for the code generator. Furthermore, additional tuning is required to ensure termination of the resulting code in the target language. Ultimately, the current version of the predicate compiler provides a fixed execution model for predicates and sets (goal-oriented depth-first search) which might not yield the best performance for the desired application. Thus, for the time being we use our proposed solution via reflexive transitive closures, but perhaps in future versions of Isabelle/HOL, the predicate compiler will be a more convenient alternative.

From Matching Problems to Identity Problems. To decide solvability of a (generalized) matching problem $\{t_1 \gtrdot \ell_1, \ldots, t_k \gtrdot \ell_k\}$, in [1], a variant of a standard matching algorithm is used which simplifies (generalized) matching problems until they are in *solved form*, i.e., all right-hand sides ℓ_i are variables (or $\bot$ is obtained which represents a matching problem without solution).

Definition 11 (Transformation of Matching Problems). *In [1] the following transformation $\Rightarrow$ on general matching problems is defined. If $\mathcal{M}$ is a general matching problem with $\mathcal{M} = \{t \gtrdot \ell\} \uplus \mathcal{M}'$ where $\ell \notin \mathcal{V}$, then*

1. $\mathcal{M} \Rightarrow \{t_1 \gtrdot \ell_1, \ldots, t_k \gtrdot \ell_k\} \cup \mathcal{M}'$, *if* $t = f(t_1, \ldots, t_k)$ *and* $\ell = f(\ell_1, \ldots, \ell_k)$
2. $\mathcal{M} \Rightarrow \bot$, *if* $t = f(\ldots)$, $\ell = g(\ldots)$, *and* $f \neq g$
3. $\mathcal{M} \Rightarrow \bot$, *if* $t \in \mathcal{V} \setminus \mathcal{V}_{\mathsf{incr}}$
4. $\mathcal{M} \Rightarrow \{t'\mu \gtrdot \ell' \mid t' \gtrdot \ell' \in \mathcal{M}\}$, *if* $t \in \mathcal{V}_{\mathsf{incr}}$

The first two rules are the standard decomposition and clash rules. Moreover, there are two special rules to handle the case where t is a variable. Here, the set of *increasing variables* $\mathcal{V}_{\mathsf{incr}} = \{x \mid \exists n.\, x\mu^n \notin \mathcal{V}\}$ plays a crucial role. It collects all those variables for which μ, if applied often enough, introduces a non-variable term. In other words, $x\mu^n$ will always be a variable for $x \notin \mathcal{V}_{\mathsf{incr}}$.

In our development, instead of using the above relation, we formalized the rules directly as a function *simplify-mp* applying the transformation rules deterministically (thereby avoiding the need for a confluence proof, as was required in [1]). As input it takes two generalized matching problems (represented by lists) where the second problem is assumed to be in solved form. Here, [] and $\cdot$ are the list constructors, and @ denotes list concatenation. The possibility of failure is encoded using Isabelle/HOL's option type, which is either *None*, in case of an error, or *Some* r for the result r. In contrast to Definition 11 of [1], our algorithm also returns an integer i which provides a lower bound on how often μ has to be applied to get a solution. The function is given by the following equations (where for brevity do-notation in the option-monad is used):

```
$simplify\text{-}mp$ [] $s = \textsf{return}\ (s, 0)$
$simplify\text{-}mp\ ((t,x) \cdot mp)$ $s = simplify\text{-}mp\ mp\ ((t,x) \cdot s)$
$simplify\text{-}mp\ ((f(ss), g(ts)) \cdot mp)$ $s = \textsf{do}\ \{\ guard\ (f = g); ps \leftarrow zip\text{-}option\ ss\ ts;$
        $simplify\text{-}mp\ (ps@mp)\ s\ \}$

$simplify\text{-}mp\ ((x, g(ts)) \cdot mp)\ s = \textsf{do}\ \{\ guard\ (x \in \mathcal{V}_{\mathsf{incr}});$
        $(mp', i) \leftarrow simplify\text{-}mp$
            $(map\text{-}\mu\ ((x, g(ts)) \cdot mp))\ (map\text{-}\mu\ s);$
        $\textsf{return}\ (mp', i+1)\ \}$
```

where, $map\text{-}\mu = map\ (\lambda(t,\ell).(t\mu,\ell))$ using the standard map function for lists, *zip-option* combines two lists of equal length into *Some* list of pairs and yields *None* otherwise, and *guard* aborts with *None* if the given predicate is not satisfied.

Example 12. For $\ell = \mathsf{eq}(x,x)$, only one of the redex problems of Example 9 remains (all others are simplified to *None*), namely $\mathsf{eq}(x,y) \gtrdot \mathsf{eq}(x,x)$, for which we obtain the simplified matching problem $\{x \gtrdot x, y \gtrdot x\}$.

In our formalization we show all relevant properties of *simplify-mp*, i.e., termination, preservation of solvability, and that *simplify-mp mp* [], if successful, is in solved form. Moreover, we prove the computed lower bound to be sound.

Theorem 13. *The function simplify-mp satisfies the following properties:*

- *It is terminating.*
- *It is complete, i.e., if (n,σ) is a solution for mp then simplify-mp mp* [] = *Some (mp', i), $i \leqslant n$, and $(n-i,\sigma)$ is a solution for mp';*
- *It is sound, i.e., if (n,σ) is a solution for mp' and simplify-mp mp* [] = *Some (mp', i) then $(n+i,\sigma)$ is a solution for mp;*
- *If simplify-mp mp* [] = *Some (mp', i) then mp' is in solved form.*

Proof. For termination of *simplify-mp mp s*, where $mp = [(t_1,\ell_1),\ldots,(t_k,\ell_k)]$, we use the lexicographic combination of the following two measures: first, we measure the sum of the sizes of the ℓ_i; and second, we measure the sum of the distances of the t_i before turning into non-variables. Here, the distance of some term t_i before turning into a non-variable is 0 if $t_i \in \mathcal{V} \setminus \mathcal{V}_{\mathsf{incr}}$ and the least number d such that $t_i\mu^d \notin \mathcal{V}$, otherwise.

For this lexicographic measure, we get a decrease in the first component for the first and the second recursive call, and a decrease in the second component for the third recursive call.

Proving soundness and completeness is done via the following property which is proven by induction on the call structure of *simplify-mp*.

Whenever *simplify-mp mp s* = r then

- if $r = None$ then $mp \cup s$ is not solvable,
- if $r = Some\ (mp', i)$, there is no solution (n,σ) for $mp \cup s$ where $n < i$, and (n,σ) is a solution for mp' iff $(n+i,\sigma)$ is a solution for $mp \cup s$.

Finally, the fact that *simplify-mp mp* [] is in solved form is shown by an easy induction proof on the call structure of *simplify-mp*, where [] is generalized to an arbitrary generalized matching problem that is in solved form. □

Although *simplify-mp* is defined as a recursive function, it cannot directly be used as a certification algorithm, due to the following two problems:

The first problem is that $\mathcal{V}_{\mathsf{incr}}$ is not executable, since it contains an existential statement (remember that we had a similar problem for $\mathcal{W}$ earlier). Again, $\mathcal{V}_{\mathsf{incr}}$ could be computed via a fixpoint computation accompanied by a tedious manual termination proof. Instead, we once more employ reflexive transitive closures to characterize $\mathcal{V}_{\mathsf{incr}}$, which allows us to use the algorithm of [12] to compute it.

Lemma 14. *Let* $R = \{(x, y) \mid x \neq y, x = y\mu, x \in \mathcal{V}, y \in \mathcal{V}\}$*. Then* $\mathcal{V}_{\mathsf{incr}} = \{y \mid \exists x \in \mathcal{V}, x\mu \notin \mathcal{V}, (x, y) \in R^*\}$*.*

The second problem is the usage of implicit parameters. Recall that at the end of Sect. 4 we just fixed some substitution μ (which corresponds to what we did in our formalization using Isabelle/HOL's locale mechanism). Obviously, both $\mathcal{V}_{\mathsf{incr}}$ and *simplify-mp* depend on μ. Hence, we have to pass μ as argument to both. As a result, the modified version of the last equation of *simplify-mp* looks as follows:

$$\begin{aligned}&\mathit{simplify\text{-}mp}\ \mu\ ((x, g(ts)) \cdot mp)\ s = \textsf{do}\ \{\mathit{guard}\ (x \in \mathcal{V}_{\mathsf{incr}}(\mu));\\ &\quad (mp', i) \leftarrow \mathit{simplify\text{-}mp}\ \mu\ (\mathit{map\text{-}\mu}\ ((x, g(ts)) \cdot mp))\ (\mathit{map\text{-}\mu}\ s);\\ &\quad \textsf{return}\ (mp', i + 1)\ \}\end{aligned} \tag{7}$$

The problem of equation (7) is its inefficiency: In every recursive call, the set of increasing variables $\mathcal{V}_{\mathsf{incr}}(\mu)$ is newly computed. Therefore, the obvious idea is to compute $\mathcal{V}_{\mathsf{incr}}(\mu)$ once and for all and pass it as an additional argument V.

$$\begin{aligned}&\mathit{simplify\text{-}mp}\ \mu\ V\ ((x, g(ts)) \cdot mp)\ s = \textsf{do}\ \{\mathit{guard}\ (x \in V);\\ &\quad (mp', i) \leftarrow \mathit{simplify\text{-}mp}\ \mu\ V\ (\mathit{map\text{-}\mu}\ ((x, g(ts)) \cdot mp))\ (\mathit{map\text{-}\mu}\ s);\\ &\quad \textsf{return}\ (mp', i + 1)\ \}\end{aligned} \tag{8}$$

This version does not have the problem of recomputing $\mathcal{V}_{\mathsf{incr}}(\mu)$ and we just have to replace the initial call *simplify-mp* μ *mp* [] by *simplify-mp* μ $\mathcal{V}_{\mathsf{incr}}(\mu)$ *mp* [].

Although, this looks straightforward and maybe not even worth mentioning, we stress that this solution does not work properly. The problem is that by introducing V, we can call *simplify-mp* using some $V \neq \mathcal{V}_{\mathsf{incr}}(\mu)$, which can cause nontermination. Take for example μ as the empty substitution and $V = \{x\}$, then the function call *simplify-mp* μ V $[(x, g(ts))]$ [] directly leads to exactly the same function call via (8). Hence, termination of *simplify-mp* defined by (8) cannot be proven. Therefore, Isabelle/HOL's function package [14] weakens equality (8) by the assumption that *simplify-mp* has to be terminating on the arguments μ, V, $((x, g(ts) \cdot mp)$, and s.

Of course, we can instantiate (8) by $V = \mathcal{V}_{\mathsf{incr}}(\mu)$. Then we can get rid of the additional assumption. But still, the corresponding unconditional equation is not suitable for code generation, since $\mathcal{V}_{\mathsf{incr}}$ on the left-hand side is not a constructor.

Our final solution is to use the recent *partial-function* [15] command of Isabelle/HOL which generates unconditional equations even for nonterminating functions, provided that some syntactic restrictions are met (only one defining equation and the function must either return an option type or be tail-recursive).

Since *simplify-mp* already returns an option type, we just had to merge all equations into a single case statement. (If the result is not of option type, we can just wrap the original return type into an option type). Afterwards the *partial-function* command is applicable and we obtain an equation similar to (8) which can be processed by the code generator and efficiently computes *simplify-mp* without recomputing $\mathcal{V}_{\mathsf{incr}}(\mu)$. Moreover, since we have already shown termination of the inefficient version of *simplify-mp*, we know that also the efficient version does terminate whenever it is called with $V = \mathcal{V}_{\mathsf{incr}}(\mu)$. In our formalization we actually have two versions of *simplify-mp*: an abstract version which is unsuitable for code generation (and also inefficient) and a concrete version. All the above properties are proven on the abstract version neglecting any efficiency problems. Afterwards it is shown that the concrete version computes the same results as the abstract one (which is relatively easy since the call-structure is the same). In this way, we get the best of two worlds: abstraction and ease of reasoning from the abstract version (using sets, existential statements, and the induction rules from the function package), and efficiency from the concrete version (using lists and concrete functions to obtain witnesses).

The above mentioned problem is not restricted to *simplify-mp*. Whenever the termination of a function relies on the correct initialization of some precomputed values, a similar problem arises. Currently, this can be solved by writing a second function via the *partial-function* command, as shown above. Although the second definition is mainly a copy of the original one, we can currently not recommend to use it as a replacement, since the function package provides much more convenience for standard definitions than when using the *partial-function* command. If the functionality of partial functions is extended, the situation might change (and we would welcome any effort in that direction).

Continuing with deciding matching problems, we are in the situation, that by using *simplify-mp* we can either directly detect that a matching problem is unsolvable or obtain an equivalent generalized matching problem in solved form $\mathcal{M} = \{t_1 \gtrdot x_1, \ldots, t_k \gtrdot x_k\}$. In principle, $\mathcal{M}$ has the solution (n, σ) where n is arbitrary and $\sigma(x_i) = t_i\mu^n$. However, this definition of σ is not always well-defined if there are i and j such that $x_i = x_j$ and $i \neq j$. To decide whether it is possible to adapt the proposed solution, we must know whether $t_i\mu^n = t_j\mu^n$ for some n, i.e., we must solve the identify problem $t_i \approxeq t_j$.

The following result of [1, Theorem 14 (iv)] is easily formalized and also poses no challenges for certification. Afterwards it remains to decide identity problems.

Theorem 15. *Let $\mathcal{M} = \{t_1 \gtrdot x_1, \ldots, t_k \gtrdot x_k\}$ be a generalized matching problem in solved form. Define $\mathcal{I}_{init} = \{t_i \approxeq t_j \mid 1 \leqslant i < j \leqslant k, x_i = x_j\}$. Then $\mathcal{M}$ is solvable iff all identity problems in $\mathcal{I}_{init}$ are solvable.*

To prove this theorem, the key observation is that we can always combine several solutions of identity problems: Whenever n_{ij} are solutions to the identity problems $t_i \approxeq t_j$, respectively, then the maximum n of all n_{ij} is a solution to all identity problems $t_i \approxeq t_j$. And then also (n, σ) is a solution to $\mathcal{M}$ where $\sigma(x_i) = t_i\mu^n$ is guaranteed to be well-defined.

Example 16. For the remaining matching problem of Example 12 we generate one identity problem: $x \approxeq y$.

Deciding Identity Problems. In [1, Section 3.4] a complicated algorithm is presented to decide solvability of an identity problem $s \approxeq t$. The main idea is to iteratively generate (s, t), $(s\mu, t\mu)$, $(s\mu^2, t\mu^2), \ldots$ until either some $(s\mu^i, t\mu^i)$ with $s\mu^i = t\mu^i$ is generated, or it can be detected that no solution exists. For the latter, some easy conditions for unsolvability are identified, e.g., $s\mu^i = C[f(ss)]$ and $t\mu^i = C[x]$ where $x \notin \mathcal{V}_{\mathsf{incr}}$. However, these conditions do not suffice to detect all unsolvable identity problems. Therefore, in each iteration conflicts (indicating which subterms have to become equal after applying μ several times, to obtain overall equality), are stored in a set S, and two sufficient conditions on pairs of conflicts from S are presented that allow to conclude unsolvability.

For the overall algorithm, soundness is rather easy to establish, completeness is more challenging, and the termination proof is the most difficult part. To be more precise, it is shown that nontermination of the algorithm allows to construct an infinite sequence of terms where no two terms are embedded into each other (which is not possible due to Kruskal's tree theorem). Hence, the formalization would require a formalization of the tree theorem. Moreover, the implicit complexity bound on the number of required iterations is quite high.

The reason for using Kruskal's tree theorem is that in [1] the conflicts in S consist of a variable, a position, and a term which is not bounded in its size. So, there is no a priori bound on S. We were able to simplify the decision procedure for $s \approxeq t$ considerably since we only store conflicts whose constituting terms are in the set of *conflict terms*

$$\mathcal{CT}(s, t) = \{u \mid v \trianglerighteq u, v \in \{s, t\} \cup ran(\mu)\}.$$

To be more precise, all conflicts are of the form (u, v, m) where (u, v) is contained in the finite set $\mathcal{S} = (\mathcal{CT}(s, t) \cap \mathcal{V}) \times \mathcal{CT}(s, t)$. Whenever we see a conflict $(u, v, \_)$ for the second time, the algorithm stops. Thus, we get a decision procedure which needs at most $|\mathcal{S}|$ iterations and whose termination proof is easy. In contrast to [1], our procedure does neither require any preprocessing on μ nor unification.

The key idea to get an a priory bound on the set of conflicts, is to consider identity problems of a generalized form $s \approxeq t\mu^n$ which can be represented by the triple (s, t, n). Then applying substitutions can be done by increasing n, and all terms that are generated during an execution of the algorithm are terms from $\mathcal{CT}(s, t)$.

Before presenting the main algorithm for deciding identity problems $s \approxeq t\mu^n$, we require an auxiliary algorithm *conflicts* (s, t, n) that computes the set of conflicts for an identity problem, i.e., subterms of s and $t\mu^n$ with different roots.

$$\begin{aligned}
&\mathit{conflicts}\ (s, y, n+1) &&= \mathit{conflicts}\ (s, \mu(y), n)\\
&\mathit{conflicts}\ (x, y, 0) &&= \textsf{if}\ x = y\ \textsf{then}\ \emptyset\ \textsf{else}\ \{(x, y, 0)\}\\
&\mathit{conflicts}\ (f(ss), y, 0) &&= \{(y, f(ss), 0)\}\\
&\mathit{conflicts}\ (x, g(ts), n) &&= \{(x, g(ts), n)\}\\
&\mathit{conflicts}\ (f(ss), g(ts), n) &&= \textsf{if}\ f = g \wedge |ss| = |ts|\\
& && \quad \textsf{then}\ \textstyle\bigcup_{(s_i,t_i) \in \mathit{zip}\ ss\ ts} \mathit{conflicts}\ (s_i, t_i, n)\\
& && \quad \textsf{else}\ \{(f(ss), g(ts), n)\}
\end{aligned}$$

We identified and formalized the following properties of *conflicts* and $\mathcal{CT}$.

Lemma 17. – *$s\sigma = t\mu^n\sigma$ iff $\forall (u, v, m) \in \mathit{conflicts}\ (s, t, n).\, u\sigma = v\mu^m\sigma$.*
– *if $(u, v, m) \in \mathit{conflicts}\ (s, t, n)$ then*
  • *$\mathit{root}(u) \neq \mathit{root}(v)$*
  • *$v \in \mathcal{V}$ implies $m = 0 \wedge u \in \mathcal{V}$*
  • *$\exists k\, p.\, n = m + k \wedge ((s|_p, t\mu^k|_p) = (u, v) \vee ((s|_p, t\mu^k|_p) = (v, u) \wedge m = 0))$*
  • *$\{u, v\} \subseteq \mathcal{CT}(s, t)$*
– *$\{u, v\} \subseteq \mathcal{CT}(s, t)$ implies $\mathcal{CT}(u, v) \subseteq \mathcal{CT}(s, t)$*
– *$\mathcal{CT}(u, v) \subseteq \mathcal{CT}(u\mu, v)$ whenever $u \in \mathcal{V}$*

Using *conflicts* we can now formulate the algorithm *ident-solve* which decides identity problem $s \approxeq t$ if invoked with *ident-solve* $\emptyset$ $(s, t, 0)$.

```
$\mathit{ident\text{-}solve}\ S\ \mathit{idp} =$
  $\textsf{let}\ \mathcal{C} = \mathit{conflicts}\ \mathit{idp}\ \textsf{in}$
  $\textsf{if}\ (f(us), \_, \_) \in \mathcal{C}\ \vee ((u, v, \_) \in \mathcal{C} \wedge (u, v, \_) \in S)\ \textsf{then}\ \mathit{None}\ \textsf{else do}\ \{$
    $\mathit{ns} \leftarrow \mathit{map\text{-}option}\ (\lambda(u, v, m).\, \mathit{ident\text{-}solve}\ (\{(u, v, m)\} \cup S)\ (u\mu, v, m+1))\ \mathcal{C};$
    $\textsf{return}\ (\mathit{max}\ \{n + 1 \mid n \in \mathit{ns}\})\ \}$
```

where *map-option* is a variant of the map function on lists whose overall result is *None* if the supplied function returns *None* for any element of the given list.

Example 18. We continue Example 16 by invoking *ident-solve* $\emptyset$ $(x, y, 0)$. This leads to the conflict $(x, y, 0)$. Afterwards, *ident-solve* $\{(x, y, 0)\}$ $(\mathsf{cons}(z, y), y, 1)$ is invoked which results in the conflict $(y, x, 0)$. Finally, the conflict $(x, y, 0)$ is generated again when calling *ident-solve* $\{(x, y, 0), (y, x, 0)\}$ $(\mathsf{cons}(z, x), x, 1)$ and the result *None* is obtained.

We formalized termination, soundness, and completeness of *ident-solve.*

Lemma 19 (Termination). *ident-solve is terminating.*

Proof. Take the measure function $\lambda S\ (s, t, \_).\, |(\mathcal{CT}(s, t) \cap \mathcal{V}) \times \mathcal{CT}(s, t) \setminus \{(a, b) \mid (a, b, \_) \in S\}|$. Then the actual termination proof boils down to showing

$$\begin{aligned}
L &:= (\mathcal{CT}(s, t) \cap \mathcal{V}) \times \mathcal{CT}(s, t) \setminus \{(a, b) \mid (a, b, \_) \in S\}\\
&\supset (\mathcal{CT}(u\mu, v) \cap \mathcal{V}) \times \mathcal{CT}(u\mu, v) \setminus \{(a, b) \mid (a, b, \_) \in \{(u, v, m)\} \cup S\} =: R
\end{aligned}$$

whenever *ident-solve* S (s, t, n) leads to a recursive call *ident-solve* $(\{(u, v, m)\} \cup S)$ $(u\mu, v, m + 1)$, i.e., whenever $(u, v, m) \in \mathit{conflicts}\ (s, t, n)$, $(u, v, \_) \notin S$, and $u \in \mathcal{V}$. By Lemma 17 we obtain $\{u, v\} \subseteq \mathcal{CT}(s, t)$ and $\mathcal{CT}(u\mu, v) \subseteq \mathcal{CT}(u, v) \subseteq \mathcal{CT}(s, t)$. Hence, $L \supseteq R$ and since $(u, v) \in L \setminus R$ we even have $L \supset R$. ◻

Lemma 20 (Soundness). *If ident-solve S $(s, t, n) =$ Some i then $s\mu^i = t\mu^n\mu^i$.*

Proof. We perform induction on the call-structure of *ident-solve*. So, assume *ident-solve* S $(s, t, n) =$ *Some* i. By definition of *ident-solve* we know that for all $(u, v, m) \in$ *conflicts* (s, t, n) there is some j such that *ident-solve* $(\{(u, v, m)\} \cup S)$ $(u\mu, v, m + 1) =$ *Some* j and i is the maximum of all $j + 1$. Using the induction hypothesis, we conclude $u\mu^{j+1} = u\mu\mu^j = v\mu^{m+1}\mu^j = v\mu^m\mu^{j+1}$ for all $(u, v, m) \in$ *conflicts* (s, t, n), and since $i \geq j + 1$ we also achieve $u\mu^i = v\mu^m\mu^i$. But this is equivalent to $s\mu^i = t\mu^n\mu^i$ by Lemma 17 (where $\sigma = \mu^i$). □

Lemma 21 (Completeness). *Whenever the identity problem $s \approxeq t$ is solvable then ident-solve $\emptyset$ $(s, t, 0) \neq$ None.*

Proof. If $s \approxeq t$ is solvable then there is some N such that $s\mu^N = t\mu^N$. Our actual proof shows the following property ($\star$) for all S, s', t', n, n', and p where $(a, b) \overset{\leftrightarrow}{=} (c, d)$ abbreviates $(a, b) = (c, d) \vee (a, b) = (d, c)$.[1]

$$(s\mu^n|_p, t\mu^n|_p) \overset{\leftrightarrow}{=} (s', t'\mu^{n'}) \tag{9}$$

$$\longrightarrow (\forall (u, v, m) \in S. (m = 0 \vee v \notin \mathcal{V}) \wedge \mathit{root}(u) \neq \mathit{root}(v) \wedge \tag{10}$$

$$(\exists q_1\, q_2\, n_1. p = q_1 q_2 \wedge n_1 < n \wedge (s\mu^{n_1}|_{q_1}, t\mu^{n_1}|_{q_1}) \overset{\leftrightarrow}{=} (u, v\mu^m)))$$

$$\longrightarrow \textit{ident-solve}\ S\ (s', t', n') \neq \textit{None} \tag{11}$$

Once ($\star$) is established, the lemma immediately follows from ($\star$) which is instantiated by $S = \emptyset$, $s' = s$, $t' = t$, $n' = n = 0$, and $p = \epsilon$ (the empty position).

To prove ($\star$), we perform induction on the call-structure of *ident-solve*. So, we assume (9) and (10), and have to show (11). By $s\mu^N = t\mu^N$ we conclude $s\mu^n|_p\mu^N = s\mu^N\mu^n|_p = t\mu^N\mu^n|_p = t\mu^n|_p\mu^N$, and thus $s'\mu^N = t'\mu^{n'}\mu^N$ by (9). By Lemma 17 this shows $u\mu^N = v\mu^m\mu^N$ for all $(u, v, m) \in$ *conflicts* $(s', t', n') =: \mathcal{C}$. In a similar way we prove $u\mu^N = v\mu^m\mu^N$ for all $(u, v, m) \in S$ using (10).

Next we consider an arbitrary $(u, v, m) \in \mathcal{C}$. By Lemma 17 we have $\mathit{root}(u) \neq \mathit{root}(v)$, $m = 0 \vee v \notin \mathcal{V}$, and there are q_1 and k such that $n' = m + k$ and $(s'|_{q_1}, t'\mu^k|_{q_1}) = (u, v) \vee ((s'|_{q_1}, t'\mu^k|_{q_1}) = (v, u) \wedge m = 0)$. In particular, this implies $(s'|_{q_1}, t'\mu^k|_{q_1}\mu^m) \overset{\leftrightarrow}{=} (u, v\mu^m)$. Moreover, we know $u\mu^N = v\mu^m\mu^N$.

First, we show that $u \in \mathcal{V}$, and hence the condition $(f(us), \_, \_) \in \mathcal{C}$ is not satisfied. The reason is that $u \notin \mathcal{V}$ also implies $v \notin \mathcal{V}$ by Lemma 17 which implies the contradiction $\mathit{root}(u) = \mathit{root}(u\mu^N) = \mathit{root}(v\mu^m\mu^N) = \mathit{root}(v) \neq \mathit{root}(u)$.

Second, *ident-solve* $(\{(u, v, m)\} \cup S)$ $(u\mu, v, m + 1) \neq$ *None*. To show this, we just apply the induction hypothesis where it remains to show that (9) and (10) are satisfied (where the values of S, s' ,t' ,n, n', p are $\{(u, v, m)\} \cup S$, $u\mu$, v, $n+1$, $m + 1$, and pq_1, respectively). To this end, we derive the following equality.

$$\begin{aligned}(s\mu^n|_{pq_1}, t\mu^n|_{pq_1}) &= (s\mu^n|_p|_{q_1}, t\mu^n|_p|_{q_1}) \overset{\leftrightarrow}{=} (s'|_{q_1}, t'\mu^{n'}|_{q_1}) \\ &= (s'|_{q_1}, t'\mu^k|_{q_1}\mu^m) \quad \overset{\leftrightarrow}{=} (u, v\mu^m).\end{aligned} \tag{12}$$

[1] In the formalization, ($\star$) looks even more complicated, since here we dropped all parts that restrict p and q_1 to valid positions.

Using (12), $root(u) \neq root(v)$, and $m = 0 \vee v \notin \mathcal{V}$, we conclude that (10) is satisfied for the new conflict (u, v, m). Moreover, (10) is trivially satisfied for all (old) conflicts in S, by using (10) (for the old inputs $(s', t', n'), \ldots$). Finally, by applying μ on all terms in (12) we obtain $(s\mu^{n+1}|_{pq_1}, t\mu^{n+1}|_{pq_1}) \overset{\leftrightarrow}{=} (u\mu, v\mu^{m+1})$ which is exactly the required (9).

The last potential reason for *ident-solve* S (s', t', n') to be *None*, is that there is some m' such that $(u, v, m') \in S$. We assume that such an m' exists and eventually show a contradiction (the most difficult part of this proof). By (10) we conclude that $m' = 0 \vee v \notin \mathcal{V}$ and there are p_1, q_3, and n_2 where $p = p_1q_3$, $n_2 < n$, and $(s\mu^{n_2}|_{p_1}, t\mu^{n_2}|_{p_1}) \overset{\leftrightarrow}{=} (u, v\mu^{m'})$. Since $n_2 < n$ there is some k_1 with $n = n_2 + k_1$ and $k_1 > 0$. Starting from (12) we derive

$$\begin{aligned} &(u, v\mu^m) && \overset{\leftrightarrow}{=} (s\mu^n|_{pq_1}, t\mu^n|_{pq_1}) \\ = &(s\mu^{n_2+k_1}|_{p_1q_3q_1}, t\mu^{n_2+k_1}|_{p_1q_3q_1}) && = (s\mu^{n_2}|_{p_1}\sigma|_q, t\mu^{n_2}|_{p_1}\sigma|_q) \\ \overset{\leftrightarrow}{=} &(u\sigma|_q, v\mu^{m'}\sigma|_q) \end{aligned} \tag{13}$$

where σ and q are abbreviations for μ^{k_1} and q_3q_1, respectively. Using (13) it is possible to derive a contradiction via a case analysis.

If $m' = m$ then (13) yields both $u\sigma\sigma|_{qq} = u$ and $v\mu^m\sigma\sigma|_{qq} = v\mu^m$. Thus, $u(\sigma\sigma)^i|_{(qq)^i} = u$ and $v\mu^m(\sigma\sigma)^i|_{(qq)^i} = v\mu^m$ for all i. For $m' = m$ we can further show $u \neq v\mu^m$ and hence, $u(\sigma\sigma)^i|_{(qq)^i} \neq v\mu^m(\sigma\sigma)^i|_{(qq)^i}$ for all i. This leads to the desired contradiction since we know that $u\mu^N = v\mu^m\mu^N$, and hence $u(\sigma\sigma)^N = u\mu^{2k_1N} = u\mu^N\mu^{(2k_1-1)N} = v\mu^m\mu^N\mu^{(2k_1-1)N} = v\mu^m\mu^{2k_1N} = v\mu^m(\sigma\sigma)^N$, which shows that for $i = N$ the previous inequality does not hold.

Otherwise $m \neq m'$. Hence, $m \neq 0 \vee m' \neq 0$ and in combination with $m = 0 \vee v \notin \mathcal{V}$ and $m' = 0 \vee v \notin \mathcal{V}$ we conclude $v \notin \mathcal{V}$. Thus, $u \in \mathcal{V}$ by Lemma 17 as $(u, v, m) \in$ *conflicts* (s', t', n'). Then by a case analysis on (13) we can show that there are i and j such that $u\mu^i \rhd u\mu^j$. Moreover, from $u\mu^N = v\mu^m\mu^N$ and $u\mu^N = v\mu^{m'}\mu^N$ we obtain $u\mu^{N+m} = u\mu^{N+m'}$. In combination with $m \neq m'$ and $u\mu^i \rhd u\mu^j$ this leads to the desired contradiction. ☐

Putting all lemmas on *ident-solve* together, we can even give a decision procedure for identity problems which does not require *ident-solve* at all, and shows an explicit bound on a solution.

Theorem 22. *An identity problem* $s \approxeq t$ *is solvable iff* $s\mu^n = t\mu^n$ *where* $n = |\mathcal{CT}(s,t) \cap \mathcal{V}| \cdot |\mathcal{CT}(s,t)|$.

Proof. If an identity problem is solvable, then the result of *ident-solve* $\emptyset$ $(s, t, 0) =$ *Some* i for some i by Lemma 21. From the termination proof in Lemma 19 we know that $i \leqslant |\mathcal{CT}(s,t) \cap \mathcal{V}| \cdot |\mathcal{CT}(s,t)| = n$ (unfortunately, in Isabelle/HOL we could not extract this knowledge from the termination proof and had to formalize this simple result separately). And by Lemma 20 we infer that $s\mu^i = t\mu^i$. But then also $s\mu^n = t\mu^n$. ☐

Note that $|\mathcal{CT}(s,t)| \leqslant |s| + |t| + |\mu|$ where $|\mu|$ is the size of all terms in the range of μ. Hence, the value of n in Theorem 22 is quadratic in the size of the input problem. We conjecture that even a linear bound exists, although some proof attempts failed. As an example, we tried to replace the condition $(u, v, \_) \in \mathcal{C} \wedge (u, v, \_) \in S$ by $(u, \_, \_) \in \mathcal{C} \wedge (u, \_, \_) \in S$ in *ident-solve* to get a linear number of iterations. However, then *ident-solve* is not complete anymore.

6 Conclusions

We have formalized several techniques to certify compositional (innermost) nontermination proofs, where the hardest part was the decision procedure of [1], which decides whether a loop is an innermost loop. In our formalization, we were able to simplify the algorithm and the proofs for identity problems considerably: a complex algorithm can be replaced by a single line due to Theorem 22.

With this result we can also show (but have not formalized) that all considered decision problems of this paper are in P.

Theorem 23. *Deciding whether an identity problem, a matching problem, or a redex problem is solvable is in P. Moreover, deciding whether a loop is an innermost loop is in P.*

Proof. We start with identity problems. By Theorem 22 we just have to check $s\mu^n = t\mu^n$ for $n = |\mathcal{CT}(s,t) \cap \mathcal{V}| \cdot |\mathcal{CT}(s,t)|$. When using DAG compressed terms we can represent $s\mu^n$ and $t\mu^n$ in polynomial space and in turn use the algorithms of [16,17] to check equality in polynomial time. Note that even if the input (s, t, μ) is already DAG compressed, the problem is still in P. The reason is that $|\mathcal{CT}(s,t)| \leqslant |s| + |t| + |\mu|$ also holds when sizes of terms are measured according to their DAG representation.

For matching problems $t \gtrdot \ell$, we first observe that *simplify-mp* $[(t, \ell)]$ $[]$ requires at most $|\mathcal{V}_{\mathsf{incr}}| \cdot |\ell|$ many iterations, and when using DAG compression, the resulting simplified matching problem can be represented in polynomial space. Hence, the resulting identity problems can all be solved in polynomial time.

Using the result for matching problems, by Theorem 8 it follows that redex problems $t \mathrel{|\!\gtrdot} \ell$ are decidable in P: The number of matching problems in $\mathcal{M}_{init}$ as well as the size of each element of $\mathcal{M}_{init}$ is linear in the sizes of t, ℓ, and μ.

Finally, since redex problems can be decided in P, by Lemma 5 this also holds for the question, whether a loop is an innermost loop. $\square$

We have also shown how reflexive transitive closures can be used to avoid termination proofs, and how partial functions help to develop efficient algorithms.

We tested our algorithms within our certifier CeTA (version 2.3) in combination with the termination analyzer AProVE [18], which is (as far as we know) currently the only tool, that can prove innermost nontermination of term rewrite systems. Through our experiments, a major soundness bug in AProVE was revealed: one of the two loop-finding methods completely ignored the strategy. After this bug was fixed, all generated nontermination proofs could be certified. Since the overhead for certification is negligible (AProVE required 151 minutes to generate all proofs, whereas CeTA required 4 seconds to certify them), we encourage termination tool users to always certify their proofs. For more details on the experiments, we refer to http://cl-informatik.uibk.ac.at/software/ceta/experiments/nonterm/.

Future work consists of integrating further techniques for which completeness is not obvious into our framework. Examples are innermost narrowing [10] and the switch from innermost termination to termination for TRSs and DPPs.

Acknowledgments. We thank Lukas Bulwahn for helpful information on Isabelle/HOL's predicate compiler.

References

1. Thiemann, R., Giesl, J., Schneider-Kamp, P.: Deciding Innermost Loops. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 366–380. Springer, Heidelberg (2008), doi:10.1007/978-3-540-70590-1_25
2. 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), doi:10.1007/978-3-642-12251-4_9
3. 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), doi:10.1007/978-3-642-03359-9_31
4. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002), doi:10.1007/3-540-45949-9
5. Baader, F., Nipkow, T.: Term Rewriting and *All That*., Paperback edn. Cambridge University Press, New York (1999), doi:10.2277/0521779200
6. Ben Cherifa, A., Lescanne, P.: Termination of rewriting systems by polynomial interpretations and its implementation. Sci. Comput. Program. 9(2), 137–159 (1987), doi:10.1016/0167-6423(87)90030-X
7. Lankford, D.S.: On proving term rewriting systems are Noetherian. Memo MTP-3, Louisiana Technical University, Ruston, LA, USA (May 1979)
8. Zantema, H.: Termination of string rewriting proved automatically. J. Autom. Reasoning 34(2), 105–139 (2005), doi:10.1007/s10817-005-6545-0
9. Sternagel, C., Thiemann, R.: Signature Extensions Preserve Termination - An Alternative Proof Via Dependency Pairs. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 514–528. Springer, Heidelberg (2010), doi:10.1007/978-3-642-15205-4_39
10. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theor. Comput. Sci. 236(1-2), 133–178 (2000),doi:10.1016/S0304-3975(99)00207-8
11. Zankl, H., Sternagel, C., Hofbauer, D., Middeldorp, A.: Finding and Certifying Loops. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901, pp. 755–766. Springer, Heidelberg (2010), doi:10.1007/978-3-642-11266-9_63
12. Sternagel, C., Thiemann, R.: Executable Transitive Closures of Finite Relations. In: The Archive of Formal Proofs (March 2011), http://afp.sf.net/entries/Transitive-Closure.shtml, Formalization
13. 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), doi:10.1007/978-3-642-03359-9_11

14. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning 44(4), 303–336 (2010), doi:10.1007/s10817-009-9157-2
15. Krauss, A.: Recursive definitions of monadic functions. In: PAR 2010. EPTCS, vol. 43, pp. 1–13 (2010), doi:10.4204/EPTCS.43.1
16. Busatto, G., Lohrey, M., Maneth, S.: Efficient Memory Representation of XML Documents. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 199–216. Springer, Heidelberg (2005), doi:10.1007/11601524_13
17. Schmidt-Schauß, M.: Polynomial equality testing for terms with shared substructures. Frank report 21, Institut für Informatik. FB Informatik und Mathematik. J.W. Goethe-Universität, Frankfurt am Main (2005)
18. 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), doi:10.1007/11814771_24

A Compact Proof of Decidability for Regular Expression Equivalence

Andrea Asperti

Department of Computer Science, University of Bologna
asperti@cs.unibo.it

Abstract. The article describes a *compact* formalization of the relation between regular expressions and deterministic finite automata, and a formally verified, *efficient* algorithm for testing regular expression equivalence, both based on the notion of *pointed regular expression* [8].

1 Introduction

In this paper, we give a simple formalization of the construction of a deterministic finite automaton associated with a given regular expression, and of a bisimilarity algorithm to check regular expression equivalence. Our approach is based on the notion of *pointed regular expression* (pre), introduced in [8] (a similar notion has been independently presented in [14]). A pointed regular expression is just a regular expression internally labelled with some additional points. Intuitively, points mark the positions inside the regular expression which have been reached after reading some prefix of the input string, or better the positions where the processing of the remaining string has to be started. Each pointed expression for e represents a state of the *deterministic* automaton associated with e; since we obviously have only a finite number of possible labellings, the number of states of the automaton is finite.

Pointed regular expressions provide the tool for an *algebraic* revisitation of McNaughton and Yamada's algorithm for position automata [19], making the proof of its correctness, that is far from trivial (see e.g. [9,11,12]), particularly clear and simple. In particular, pointed expressions offer an appealing alternative to Brzozowski's derivatives (see e.g. [20] for a recent revisitation), avoiding their weakest point, namely the fact of being forced to quotient derivatives w.r.t. a suitable notion of equivalence in order to get a finite number of states (that is not essential for recognizing strings, but is crucial for comparing regular expressions).

All the proofs in this paper have been formalized in the Interactive Theorem Prover Matita [6].

2 Preliminaries

An *alphabet* is an arbitrary set of elements, equipped with a decidable equality:

```
record DeqSet : Type :=
{  carr :>Type;                  (* coercion *)
   eqb: carr → carr → bool;      (* notation:  a == b *)
   eqb_true: ∀x,y. (eqb x y = true) ↔ (x = y)
}.
```

A string (or word) over the alphabet S is just an element of *list S*. We need to deal with languages, that is, sets of strings. A traditional way to encode sets of elements in a given universe U in type theory is by means of predicates over U, namely elements of $U \to Prop$. A language over an alphabet S is hence an element of *list S → Prop.*

Languages inherit all the basic operations for sets, namely union, intersection, complementation, substraction, and so on. In addition, we may define some new operations induced by string concatenation, and in particular the *concatenation* $A \cdot B$ of two languages A and B, the so called *Kleene's star* A^* of A and the *derivative* of a language A w.r.t. a given character a:

```
definition cat :=λS,A,B.λw:word S.
  ∃w1,w2.w1 @ w2 = w ∧ A w1 ∧ B w2.

definition star :=λS,A,λw:word S.
  ∃lw. flatten  S lw = w ∧  list_forall  S lw A.

definition deriv :=λS,A,a,w. A (a::w).
```

In the definition of star, $flatten$ and $list\_forall$ are standard functions over lists, respectively mapping $[l_1,\ldots,l_n]$ to $l_1@l_2\ldots@l_n$ and $[w_1,w_2,\ldots,w_n]$ to $(A\ w_1) \wedge (A\ w_2) \cdots \wedge (A\ w_n)$.

Two languages are equal if they are equal as sets, namely if they contain the same words. This notion of equality, called `eqP` and denoted with the infix operator $\simeq$, is an extensional equality, different from the primitive intensional equality of Matita. In particular, we can *rewrite* with an equation $A \simeq B$ inside a context $C[A]$, only if the context is compatible with "$\simeq$".

```
definition eqP :=λA:Type.λP,Q:A → Prop.∀a:A.P a ↔Q a.
```

The main equations between languages that we shall need for the purposes of this paper (in addition to the set theoretic ones, and those expressing extensionality of operations) are listed below; the simple proofs are omitted.

```
lemma epsilon_cat_r: ∀S.∀A:word S → Prop. A ·{ϵ} ≃ A.
lemma epsilon_cat_l: ∀S.∀A:word S → Prop. {ϵ} ·A ≃ A.
lemma distr_cat_r: ∀S.∀A,B,C:word S → Prop. (A ∪B) ·C ≃  A · C ∪ B · C.
lemma deriv_union: ∀S,A,B,a. deriv (A ∪B) a ≃ (deriv A a) ∪ (deriv B a).
lemma deriv_cat: ∀S,A,B,a. ¬ A ϵ→ deriv (A·B) a ≃ (deriv A a) · B.
lemma star_fix_eps : ∀S.∀A:word S → Prop. A* ≃ (A − {ϵ}) · A* ∪ {ϵ}.
```

3 Regular Expressions

The type *re* of regular expressions over an alphabet S is the smallest collection of objects generated by the following constructors:

```
inductive re (S: DeqSet) : Type :=
   z: re S                  (* empty *)
 | e: re S                  (* epsilon *)
 | s: S → re S              (* symbol *)
 | c: re S → re S → re S    (* concatenation *)
 | o: re S → re S → re S    (* plus *)
 | k: re S → re S.          (* kleene's star *)
```

In Matita, similarly to most interactive provers, we provide mechanisms to let the user define his own notation for syntactic constructs, and in the rest of the paper we shall use the traditional notation for regular expressions, namely $\emptyset, \epsilon, a, e_1 \cdot e_2, e_1 + e_2, e^*$.

The language *sem r* (notation: $[\![r]\!]$) associated with the regular expression r is defined by the following function:

```
let rec sem (S : DeqSet) (r : re S) on r : word S → Prop :=
  match r with
  [ z ⇒ ∅
  | e ⇒ {ϵ}
  | s x ⇒ {[x]}
  | c r1 r2 ⇒ ⟦r1⟧ · ⟦r2⟧
  | o r1 r2 ⇒ ⟦r1⟧ ∪ ⟦r2⟧
  | k r1 ⇒ ⟦r1⟧* ].
```

4 Pointed Regular Expressions

A pointed item is a data type used to encode a *set of positions* inside a regular expression. The idea of formalizing pointers inside a data type by means of a labelled version of the data type itself is probably one of the first, major lessons learned in the formalization of the metatheory of programming languages (see e.g. [16] for a precursory application to residuals in lambda calculus). For our purposes, it is enough to mark positions preceding individual characters, so we shall have two kinds of characters $\bullet a$ (*pp a*) and a (*ps a*) according to the case a is pointed or not.

```
inductive pitem (S: DeqSet) : Type :=
   pz: pitem S
 | pe: pitem S
 | ps: S → pitem S
 | pp: S → pitem S
 | pc: pitem S → pitem S → pitem S
 | po: pitem S → pitem S → pitem S
 | pk: pitem S → pitem S.
```

A *pointed regular expression* (pre) is just a pointed item with an additional boolean, that must be understood as the possibility to have a trailing point *at the end* of the expression. As we shall see, pointed regular expressions can be understood as states of a DFA, and the boolean indicates if the state is final or not.

```
definition pre :=λS.pitem S × bool.
```

The *carrier* $|i|$ of an item i is the regular expression obtained from i by removing all the points. Similarly, the *carrier* of a pointed regular expression is the carrier of its item. The formal definition of this functions are straightforward, so we omit them. In the sequel, we shall use the same notation for functions defined over items or pres, leaving to the reader the simple disambiguation task (matita is also able to solve autonomously this kind of notational overloading).

The intuitive semantic of a point is to mark the position where we should start reading the regular expression. The language associated to a pre is the union of the languages associated with its points. Here is the straightforward definition (the question mark is an implicit parameter):

```
let rec semi (S : DeqSet) (i : pitem S) on i : word S → Prop :=
match r with
[ pz ⇒ ∅
| pe ⇒ ∅
| ps _ ⇒ ∅
| pp x ⇒ {[x]}
| pc i1 i2 ⇒ (semi ? i1) · ⟦|i2|⟧ ∪ (semi ? i2)
| po i1 r2 ⇒ (semi ? i1) ∪ (semi ? i2)
| pk i1 ⇒ (semi ? i1) · ⟦|i1|⟧^* ].

definition semp :=λS : DeqSet.λp:pre S.
  if (snd p) then semi ? (fst p) ∪ {ϵ} else semi ? (fst p).
```

In the sequel, we shall often use the same notation for functions defined over re, items or pres, leaving to the reader the simple disambiguation task (matita is also able to solve autonomously this kind of notational overloading). In particular, we shall denote with $[\![e]\!]$ all semantic functions *sem*, *semi* and *semp*.

Example 1.

1. If e contains no point then $[\![e]\!] = \emptyset$
2. $[\![(a+\bullet bb)^*]\!] = [\![bb(a+bb)^*]\!]$ □

Here are a few, simple, semantic properties of items

```
lemma not_epsilon_item : ∀S:DeqSet.∀i:pitem S. ¬ (⟦i⟧ ϵ).
lemma epsilon_pre : ∀S.∀e:pre S. (⟦i⟧ ϵ) ↔ (snd e = true).
lemma minus_eps_item: ∀S.∀i:pitem S. ⟦i⟧ ≃ ⟦i⟧−{ϵ}.
lemma minus_eps_pre: ∀S.∀e:pre S. ⟦fst e⟧ ≃ ⟦e⟧−{ϵ}.
```

The first property is proved by a simple induction on i; the other results are easy corollaries.

4.1 Intensional Equality of Pres

Items and pres are a very concrete datatype: they can be effectively compared, and enumerated. This is important, since pres *are* the states of our finite automata, and we shall need to compare states for bisimulation in Section 7.

In particular, we can define *beqitem* and *beqitem_true* enriching the set (*pitemS*) to a *DeqSet*.

```
definition DeqItem :=λS.
  mk_DeqSet (pitem S) (beqitem S) (beqitem_true S).
```

Matita's mechanism of *unification hints* [7] allows the type inference system to look at (*pitemS*) as the carrier of *DeqSet*, and at *beqitem* as if it was the equality function of *DeqSet*.

The product of two DeqSets is clearly still a DeqSet. Via unification hints, we may enrich a product type to the corresponding DeqSet; since moreover the type of booleans is a DeqSet too, this means that the type of pres *automatically inherits* the structure of a DeqSet (in Section 7, we shall deal with pairs of pres, and in this case too, without having anything to declare, the type will inherit the structure of a DeqSet).

Items and Pres can also be enumerated. In particular, it is easy to define a function *pre_enum* that takes in input a regular expression and gives back the list of all pres having e for carrier. Completeness of *pre_enum* is stated by the following lemma:

```
lemma pre_enum_complete : ∀S.∀e:pre S.
  memb ? e (pre_enum S (|fst e|)) = true.
```

5 Broadcasting Points

Intuitively, a regular expression e must be understood as a pointed expression with a single point in front of it. Since however we only allow points before symbols, we must broadcast this initial point inside e traversing all nullable subexpressions, that essentially corresponds to the ϵ-closure operation on automata. We use the notation $\bullet(\cdot)$ to denote such an operation; its definition is the expected one: let us start discussing an example.

Example 2. Let us broadcast a point inside $(a+\epsilon)(b^*a+b)b$. We start working in parallel on the first occurrence of a (where the point stops), and on ϵ that gets traversed. We have hence reached the end of $a+\epsilon$ and we must pursue broadcasting inside $(b^*a+b)b$. Again, we work in parallel on the two additive subterms b^*a and b; the first point is allowed to both enter the star, and to traverse it, stopping in front of a; the second point just stops in front of b. No point reached that end of b^*a+b hence no further propagation is possible. In conclusion:

$$\bullet((a+\epsilon)(b^*a+b)b) = \langle(\bullet a+\epsilon)((\bullet b)^* \bullet a + \bullet b)b, \mathit{false}\rangle \qquad \square$$

Broadcasting a point inside an item generates a pre, since the point could possibly reach the end of the expression. Broadcasting inside a pair $i_1 + i_2$ amounts to broadcast in parallel inside i_1 and $i2$. If we define

$$\langle i_1, b_1' \rangle \oplus \langle i_2, b_2 \rangle = \langle i_1 + i_2, b_1 \vee b_2 \rangle$$

then, we just have $\bullet(i_1 + i_2) = \bullet(i_1) \oplus \bullet(i_2)$.

Concatenation is a bit more complex. In order to broadcast an item inside $i_1 \cdot i_2$ we should start broadcasting it inside i_1 and then proceed into i_2 if and only if a point reached the end of i_1.

This suggests to define $\bullet(i_1 \cdot i_2)$ as $\bullet(i_1) \triangleright i_2$, where $e \triangleright i$ is a general operation of concatenation between a pre and item (named *pre_concat_l*) defined by cases on the boolean in e

$$\langle i_1, \mathit{true} \rangle \triangleright i_2 = i_1 \triangleleft \bullet(i_2) \qquad \langle i_1, \mathit{false} \rangle \triangleright i_2 = \langle i_1 \cdot i_2, \mathit{false} \rangle$$

In turn, $\triangleleft$ (named *pre_concat_r*) says how to concatenate an item with a pre, that is however extremely simple:

$$i_1 \triangleleft \langle i_1, b \rangle = \langle i_1 \cdot i_2, b \rangle$$

The different kinds of concatenation between items and pres are summarized in Fig. 1, where we also depict the concatenation between two pres of Section 5.3.

	item	pre
item	$i_1 \cdot i_2$	$i_1 \triangleleft e_2$ $i_1 \triangleleft \langle i_1, b \rangle := \langle i_1 \cdot i_2, b \rangle$
pre	$e_1 \triangleright i_2$ $\langle i_1, \mathit{true} \rangle \triangleright i_2 := i_1 \triangleleft \bullet(i_2)$ $\langle i_1, \mathit{false} \rangle \triangleright i_2 := \langle i_1 \cdot i_2, \mathit{false} \rangle$	$e1 \odot e_2$ $e_1 \odot \langle i_2, b \rangle :=$ let $\langle i', b' \rangle = e_1 \triangleright i_2$ in $\langle i', b \vee b' \rangle$

Fig. 1. Concatenations between items and pres and respective equations

The definition of $\bullet(\cdot)$ (*eclose*) and $\triangleright$ (*pre_concat_l*) are mutually recursive. In this situation, a viable alternative that is usually simpler to reason about, is to abstract one of the two functions with respect to the other.

```
definition pre_concat_l :=λS.λbcast:∀S.pitem S → pre S.λe1:pre S.λi2:pitem S.
  let ⟨i1,b1⟩ := e1 in
  if b1 then i1 ◁ (bcast ? i2) else ⟨i1 · i2, false⟩.

let rec eclose (S: DeqSet) (i: pitem S) on i : pre S :=
 match i with
  [ pz ⇒ ⟨pz S, false⟩
  | pe ⇒ ⟨pe S, true⟩
  | ps x ⇒ ⟨ps S x, false⟩
  | pp x ⇒ ⟨pp S x, false⟩
  | po i1 i2 ⇒ •i1 ⊕ •i2
  | pc i1 i2 ⇒ •i1 ◁ i2
  | pk i ⇒ ⟨(fst (•i))*,true⟩ ].
```

The definition of *eclose* can then be *lifted* from items to pres:

```
definition lift :=λS.λf:pitem S →pre S.λe:pre S.
  let ⟨i,b⟩ := e in ⟨fst (f i), snd (f i) ∨ b⟩.

definition preclose :=λS. lift S (eclose S).
```

By induction on the item i it is easy to prove the following result:

```
lemma erase_bullet : ∀S.∀i:pitem S. |fst (•i)| = |i|.
```

5.1 Semantics

We are now ready to state the main semantic properties of $\oplus, \triangleright, \triangleleft$ and $\bullet(-)$:

```
lemma sem_oplus: ∀S:DeqSet.∀e1,e2:pre S.
  ⟦e1 ⊕ e2⟧ ≃ ⟦e1⟧ ∪ ⟦e2⟧.

lemma sem_pre_concat_r : ∀S,i.∀e:pre S.
  ⟦i ▷ e⟧ ≃ ⟦i⟧ · ⟦|fst e|⟧ ∪ ⟦e⟧.

lemma sem_pre_concat_l : ∀S.∀e1:pre S.∀i2:pitem S.
  ⟦e1 ◁ i2⟧ ≃  ⟦e1⟧ · ⟦|i2|⟧ ∪ ⟦i2⟧.

theorem sem_bullet: ∀S:DeqSet. ∀i:pitem S.
  ⟦•i⟧ ≃ ⟦i⟧ ∪ ⟦|i|⟧.
```

The proofs of *sem_oplus* and *sem_pre_concat_r* are straightforward. For the others, we proceed as follow: we first prove the following auxiliary lemma, that assumes *sem_bullet*

```
lemma sem_pre_concat_l_aux : ∀S.∀e1:pre S.∀i2:pitem S.
   ⟦•i2⟧ ≃  ⟦i2⟧ ∪ ⟦|i2|⟧ →
     ⟦e1 ◁ i2⟧ ≃ ⟦e1⟧ · ⟦|i2|⟧ ∪ ⟦i2⟧.
```

Then, using the previous result, we prove *sem_bullet* by induction on i. Finally, *sem_pre_concat_l_aux* and *sem_bullet* give *sem_pre_concat_l*.

It is important to observe that all proofs have an algebraic flavor. Let us consider for instance the proof of *sem_pre_concat_l_aux*. Assuming $e_1 = \langle i_1, b_1\rangle$ we proceed by cases on b_1. If b_1 is false, the result is trivial; if b_1 is true, we have

$$\begin{aligned}
[\![\langle i_1, \mathit{true}\rangle \triangleleft i_2]\!] &\simeq [\![i_1]\!] \triangleright \bullet(i_2) && \text{by def. of } \triangleleft\\
&\simeq [\![i_1]\!] \cdot [\![|\mathit{fst}\ \bullet(i_2)|]\!] \cup [\![\bullet(i_2)]\!] && \text{by sem\_pre\_concat\_r}\\
&\simeq [\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] \cup [\![|i_2|]\!] && \text{by erase\_bullet and sem\_bullet}\\
&\simeq [\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by assoc. and comm.}\\
&\simeq ([\![i_1]\!] \cup \{\epsilon\}) \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by distr\_cat\_r}\\
&\simeq [\![\langle i_1, \mathit{true}\rangle]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by the semantics of pre}
\end{aligned}$$

As another example, let us consider the proof of *sem_bullet*. The proof is by induction on i; let us consider the case of $i_1 \cdot i_2$. We have:

$$\begin{aligned}
[\![\bullet(i_1 \cdot i_2)]\!] &\simeq [\![\bullet(i_1)]\!] \lhd [\![i_2]\!] && \text{by definition of } \bullet(\cdot)\\
&\simeq [\![\bullet(i_1)]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by sem\_pre\_concat\_l}\\
&\simeq ([\![i_1]\!] \cup [\![|i_1|]\!]) \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by induction hypothesis}\\
&\simeq [\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![|i_1|]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by distr\_cat\_r}\\
&\simeq ([\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!]) \cup [\![|i_1 \cdot i_2|]\!] && \text{by assoc. and comm.}\\
&\simeq [\![(i_1 \cdot i_2)]\!] \cup [\![|i_1 \cdot i_2|]\!] && \text{by definition of } [\![\_]\!]
\end{aligned}$$

5.2 Initial State

As a corollary of theorem *sem_bullet*, given a regular expression e, we can easily find an item with the same semantics of e: it is enough to get an item (*blank* e) having e as carrier and no point, and then broadcast a point in it:

$$[\![\bullet(\mathit{blank}\ e)]\!] \simeq [\![(\mathit{blank}\ e)]\!] \cup [\![e]\!] \simeq [\![e]\!]$$

The definition of *blank* is straightforward; its main properties (both proved by an easy induction on e) are the following:

```
lemma forget_blank: ∀S.∀e:re S.|blank S e| = e.
lemma sem_blank: ∀S.∀e:re S. [[blank S e]] ≃ ∅.
theorem re_embedding: ∀S.∀e:re S. [[•(blank S e)]] ≃ [[e]].
```

5.3 Lifted Operators

Plus and bullet have been already lifted from items to pres. We can now do a similar job for concatenation (⊙) and and Kleene's star (⊛).

```
definition lifted_cat :=λS:DeqSet.λe:pre S. lift S (pre_concat_l S eclose e).

definition lk :=λS:DeqSet.λe:pre S.
  let ⟨i1,b1⟩ := e in if b1 then ⟨(fst (eclose ? i1))*, true⟩ else ⟨i1*, false⟩.
```

We can easily prove the following properties:

```
lemma sem_odot: ∀S.∀e1,e2: pre S.
  [[e1 ⊙ e2]] ≃ [[e1]] · [[| fst e2|]] ∪ [[e2]].

theorem sem_ostar: ∀S.∀e:pre S.
  [[e⊛]] ≃ [[e]] · [[| fst e|]]*.
```

For example, let us look at the proof of the latter. Given $e = \langle i, b\rangle$ we proceed by cases on b. If b is false the result is trivial; if b is true we have:

$$\begin{aligned}
[\![\langle i, \mathit{true}\rangle^{\circledast}]\!] &\simeq [\![(\mathit{fst}\ \bullet(i))^*]\!] \cup \{\epsilon\} && \text{by definition of } \circledast\\
&\simeq [\![\mathit{fst}\ \bullet(i)]\!] \cdot [\![\mathit{fst}\ |\bullet(i)|]\!]^* \cup \{\epsilon\} && \text{by definition of } [\![\_]\!]\\
&\simeq [\![\mathit{fst}\ \bullet(i)]\!] \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by erase\_bullet}\\
&\simeq ([\![\bullet(i)]\!] - \{\epsilon\}) \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by minus\_eps\_pre}\\
&\simeq (([\![i]\!] \cup [\![|i|]\!]) - \{\epsilon\}) \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by sem\_bullet}\\
&\simeq (([\![i]\!] - \{\epsilon\}) \cup ([\![|i|]\!] - \{\epsilon\})) \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by distr\_minus}
\end{aligned}$$

$$
\begin{array}{ll}
\simeq([\![i]\!] \cup ([\![|i|]\!] - \{\epsilon\})) \cdot [\![|i|]\!]^* \cup \{\epsilon\} & \text{by minus\_eps\_item} \\
\simeq[\![i]\!] \cdot [\![|i|]\!]^* \cup ([\![|i|]\!] - \{\epsilon\}) \cdot [\![|i|]\!]^* \cup \{\epsilon\} & \text{by distr\_cat\_r} \\
\simeq[\![i]\!] \cdot [\![|i|]\!]^* \cup [\![|i|]\!]^* & \text{by star\_fix\_eps} \\
\simeq([\![i]\!] \cup \{\epsilon\}) \cdot [\![|i|]\!]^* & \text{by distr\_cat\_r} \\
\simeq[\![\langle i, \mathit{true}\rangle]\!] \cdot [\![|i|]\!]^* & \text{by definition of } [\![\_]\!]
\end{array}
$$

6 Moves

We now define the move operation, that corresponds to the advancement of the state in response to the processing of an input character a. The intuition is clear: we have to look at points inside e preceding the given character a, let the point traverse the character, and broadcast it. All other points must be removed.

We can give a particularly elegant definition in terms of the lifted operators of the previous section:

```
let rec move (S: DeqSet) (x:S) (E: pitem S) on E : pre S :=
 match E with
  [ pz ⇒ ⟨pz S, false ⟩
  | pe ⇒ ⟨pe S, false ⟩
  | ps y ⇒ ⟨ps S y, false ⟩
  | pp y ⇒ ⟨ps S, x == y⟩  (* the point is advanced if x==y, erased otherwise *)
  | po e1 e2 ⇒ (move ? x e1) ⊕ (move ? x e2)
  | pc e1 e2 ⇒ (move ? x e1) ⊙ (move ? x e2)
  | pk e ⇒ (move ? x e)^⊛ ].
```

Example 3. Let us consider the pre $(\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b)b$ and the two moves w.r.t. the characters a and b. For a, we have two possible positions (all other points gets erased); the innermost point stops in front of the final b, the other one broadcast inside $(b^* a + b)b$, so

$$\mathit{move}\ a\ ((\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b)b) = \langle (a + \epsilon)((\bullet b)^* \bullet a + \bullet b) \bullet b, \mathit{false}\rangle$$

For b, we have two positions too. The innermost point stops in front of the final b too, while the other point reaches the end of b^* and must go back through $b^* a$:

$$\mathit{move}\ b\ ((\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b) \bullet b) = \langle (a + \epsilon)((\bullet b)^* \bullet a + b) \bullet b, \mathit{false}\rangle \qquad \square$$

Obviously, a move does not change the carrier of the item, as one can easily prove by induction on the item

```
lemma same_carrier: ∀S:DeqSet.∀a:S.∀i:pitem S.
  | fst (move a i)| = |i |.
```

Here is our first, major result.

```
theorem move_ok: ∀S:DeqSet.∀a:S.∀i:pitem S.
  [[move a i]] ≃ deriv [[i]] a.
```

The proof is a simple induction on i. Let us see the case of concatentation:

$$\begin{array}{rll}
[\![move\ a\ (i_1 \cdot i_2)]\!] \simeq & [\![move\ a\ i_1 \odot move\ a\ i_2]\!] & \text{by def. of move} \\
\simeq & [\![move\ a\ i_1]\!] \cdot [\![|fst\ (move\ a\ i_2)|]\!] \cup [\![move\ a\ i_2]\!] & \text{by sem\_odot} \\
\simeq & [\![move\ a\ i_1]\!] \cdot [\![|i_2|]\!] \cup [\![move\ a\ i_2]\!] & \text{by same\_carrier} \\
\simeq & (deriv\ [\![i_1]\!]\ a) \cdot [\![|i_2|]\!] \cup (deriv\ [\![i_2]\!]\ a) & \text{by ind. hyp.} \\
\simeq & (deriv\ ([\![i_1]\!] \cdot [\![|i_2|]\!])\ a) \cup (deriv\ [\![i_2]\!]\ a) & \text{by deriv\_cat} \\
\simeq & deriv\ ([\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!])\ a & \text{by deriv\_union} \\
\simeq & deriv\ [\![i_1 \cdot i_2]\!]\ a & \text{by definition of } [\![\_]\!]
\end{array}$$

The move operation is generalized to strings in the obvious way:

```
let rec moves (S : DeqSet) w e on w : pre S :=
 match w with
  [ nil  ⇒ e
  | cons a tl   ⇒ moves S tl (move S a (fst e))].

lemma same_carrier_moves: ∀S:DeqSet.∀w.∀e:pre S.
  | fst (moves ? w e)| = | fst e |.

theorem decidable_sem: ∀S:DeqSet.∀w: word S. ∀e:pre S.
   (snd (moves ? w e) = true) ↔ [[e]] w.
```

The proof of *decidable_sem* is by induction on w. The case $w = \epsilon$ is trivial; if $w = a :: w_1$ we have

$$\begin{array}{lll}
snd\ (moves\ (a :: w_1)\ e) = true & & \\
\quad \leftrightarrow & snd\ (moves\ w_1\ (move\ a\ (fst\ e))) = true & \text{by def. of moves} \\
\quad \leftrightarrow & [\![move\ a\ (fst\ e)]\!]\ w_1 & \text{by ind. hyp.} \\
\quad \leftrightarrow & [\![e]\!]\ a :: w_1 & \text{by move\_ok}
\end{array}$$

It is now clear that we can build a DFA D_e for e by taking pre as states, and move as transition function; the initial state is $\bullet(e)$ and a state $\langle i, b\rangle$ is final if and only if $b = true$. The fact that states in D_e are finite is obvious: in fact, their cardinality is at most 2^{n+1} where n is the number of symbols in e. This is one of the advantages of pointed regular expressions w.r.t. derivatives, whose finite nature only holds after a suitable quotient.

Example 4. Figure 2 describes the DFA for the regular expression $(ac + bc)^*$. The graphical description of the automaton is the traditional one, with nodes for states and labelled arcs for transitions. Unreachable states are not shown.

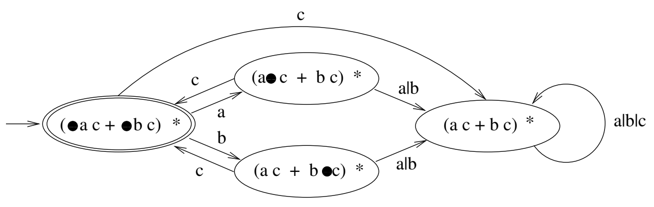


Fig. 2. DFA for $(ac + bc)^*$

Final states are emphasized by a double circle: since a state $\langle e, b\rangle$ is final if and only if b is true, we may just label nodes with the item.

The automaton is not minimal: it is easy to see that the two states corresponding to the pres $(a \bullet c + bc)^*$ and $(ac + b \bullet c)^*$ are equivalent (a way to prove it is to observe that they define the same language!). In fact, each state has a clear semantics given in terms of the associated pre e and not of the behaviour of the automaton. As a consequence, the construction of the automaton is not only *direct*, but also extremely *intuitive* and *locally verifiable*. □

Example 5. Starting from the regular expression $(a + \epsilon)(b^* a + b)b$, we obtain the automaton in Figure 3. Remarkably, this DFA is minimal, testifying the small

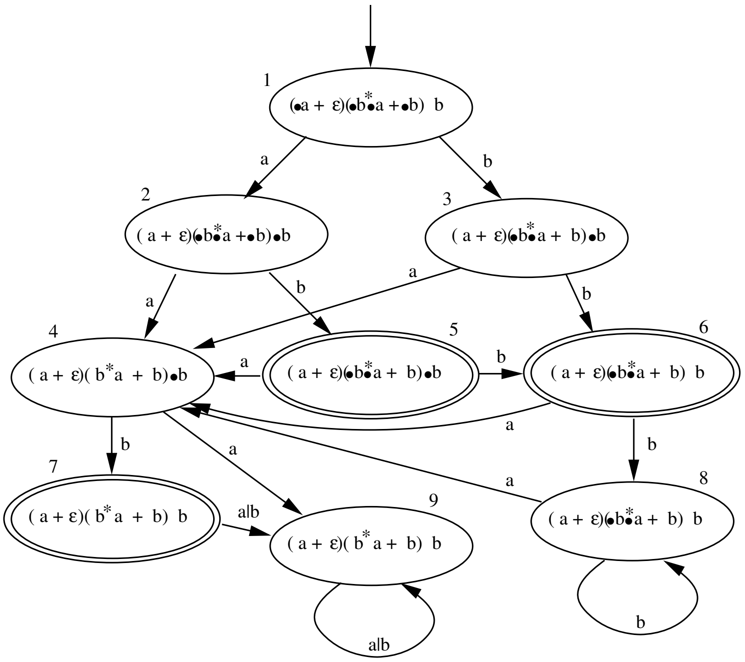


Fig. 3. DFA for $(a + \epsilon)(b^* a + b)b$

number of states produced by our technique (the pair of states $6 - 8$ and $7 - 9$ differ for the fact that 6 and 7 are final, while 8 and 9 are not). □

7 Equivalence

We say that two pres $\langle i_1, b_1\rangle$ and $\langle i_2, b_2\rangle$ are *cofinal* if and only if $b_1 = b_2$.

As a corollary of *decidable_sem*, we have that two expressions e_1 and e_2 are equivalent iff for any word w the states reachable through w are cofinal.

```
theorem equiv_sem: ∀S:DeqSet.∀e1,e2:pre S.
  ⟦e1⟧ ≃ ⟦e2⟧ ↔ ∀w.cofinal ⟨moves w e1,moves w e2⟩.
```

This does not directly imply decidability: we have no bound over the length of w; moreover, so far, we made no assumption over the cardinality of S. Instead of requiring S to be finite, we may restrict the analysis to characters occurring in the given pres. This means we can prove the following, stronger result:

```
lemma equiv_sem_occ: ∀S.∀e1,e2:pre S.(∀w.(sublist S w (occ S e1 e2))→
  cofinal ⟨moves w e1,moves w e2⟩) → ⟦e1⟧ ≃ ⟦e2⟧.
```

The proof essentially requires the notion of sink state and a few trivial properties:

```
definition sink_pre :=λS.λi.⟨blank S (| i |), false⟩.

lemma not_occur_to_sink: ∀S,a.∀i:pitem S. memb S a (occur S (|i|)) ≠ true →
  move a i = sink_pre S i.

lemma moves_sink: ∀S,w,i. moves w (sink_pre S i) = sink_pre S i.
```

Let us say that a list of pairs of pres is a *bisimulation* if it is closed w.r.t. moves, and all its members are cofinal.

```
definition sons :=λS:DeqSet.λl:list S.λp:(pre S)×(pre S).
  map ?? (λa.⟨move a (fst (fst p)),move a (fst (snd p))⟩) l.

definition is_bisim :=λS:DeqSet.λl:list ?.λalpha:list S. ∀p:(pre S)×(pre S).
  memb ? p l = true → cofinal ? p ∧ (sublist ? (sons ? alpha p) l).
```

Using lemma *equiv_sem_occ* it is easy to prove

```
lemma bisim_to_sem: ∀S:DeqSet.∀l:list ?.∀e1,e2: pre S.
  is_bisim S l (occ S e1 e2) → memb ? ⟨e1,e2⟩ l = true → ⟦e1⟧ ≃ ⟦e2⟧.
```

As observed in [18] this is already an interesting result: checking if l is a bisimulation is decidable, hence we could generate l with some untrusted piece of code and then run a (boolean version of) *is_bisim* to check that it is actually a bisimulation. However, in order to prove that equivalence of regular expressions is *decidable* we must prove that we can always effectively build such a list (or find a counterexample). The idea is that the list we are interested in is just the set of all pair of pres *reachable* from the initial pair via some sequence of moves.

The algorithm for computing reachable nodes in a graph is a very traditional one. We split nodes in two disjoint lists: a list of *visited* nodes and a *frontier*, composed by all nodes connected to a node in visited but not visited already. At each step we select a node a from the frontier, compute its sons, add a to the set of visited nodes and the (not already visited) sons to the frontier.

Instead of fist computing reachable nodes and then performing the bisimilarity test we can directly integrate it in the algorithm: the set of visited nodes is closed by construction w.r.t. reachability, so we have just to check cofinality for any node we add to visited.

Here is the extremely simple algorithm

```
let rec bisim S l n ( frontier , visited : list ?) on n :=
  match n with
  [ O ⇒ ⟨false,visited⟩ (* assert false *)
  | S m ⇒
    match frontier with
    [ nil ⇒ ⟨true,visited⟩
    | cons hd tl ⇒
      if beqb (snd (fst hd)) (snd (snd hd)) (* cofinality *) then
        bisim S l m (unique_append ? (filter ? (λx.notb (memb ? x (hd::visited)))
        (sons S l hd)) tl) (hd:: visited )
      else ⟨ false , visited ⟩
    ]
  ].
```

The integer n is an upper bound to the number of recursive calls, equal to the dimension of the graph. It returns a pair composed by a boolean and the set of visited nodes; the boolean is true if and only if all visited nodes are cofinal.

The main test function is:

```
definition equiv :=λSig.λre1,re2:re Sig.
  let e1 :=•(blank ? re1) in
  let e2 :=•(blank ? re2) in
  let n :=S (length ? (space_enum Sig (|fst e1|) (| fst e2 |))) in
  let sig :=(occ Sig e1 e2) in
  (bisim ? sig n [⟨e1,e2⟩] []).
```

We proved both correctness and completeness; in particular, we have

```
theorem euqiv_sem : ∀Sig.∀e1,e2:re Sig.
   fst (equiv ? e1 e2) = true ↔ ⟦e1⟧ ≃ ⟦e2⟧.
```

For correctness, we use the invariant that at each call of *bisim* the two lists *visited* and *frontier* only contain nodes reachable from $\langle e_1, e_2\rangle$: hence it is absurd to suppose to meet a pair which is not cofinal. For completeness, we use the invariant that all the nodes in visited are cofinal, and the sons of *visited* are either in *visited* or in the *frontier*; since at the end *frontier* is empty, *visited* is hence a bisimulation. All in all, correctness and completeness take little more than a few hundreds lines.

8 Discussion, Related Works, Conclusions

Most of the formal proofs contained in this paper go back to 2009, preceding the technical report where we introduced the notion of pointed regular expression [8]; the long term idea, still in progress, was to use this material as a base for wrting an introductory tutorial to Matita. Since then, a small bunch of related works have appeared [2,18,13,22], convincing us we could possibly add our two cents to this interesting, and apparently never exhausted topic.

Most of the above mentioned works are based on the notion of *derivative*, either in Brzozowski's acception [18,13] or in Antimirov's one [2,22]. This is not particularly surprising, since the algebraic nature of derivatives make them particularly appealing for a formal development. However, as remarked in [18], "in the large range of algorithms that turn regular expressions into automata, Brzozowski's procedure is on the elegant side, not the efficient one".

In order to get an efficient implementation, Braibant and Pous [10] resort to a careful implementation of finite state automata, encoding them as matrices over the given alphabet; automata are build using a variant of Thompson's technique [21] due to Ilie and Yu [17] (simpler to formalize then [21] but still complex).

Our approach based on pointed regular expressions provides a simple, algebraic revisitation of McNaughton and Yamada's algorithm [19] that, in contrast to Brzozowski's procedure, is traditionally reputed for its efficiency [1]; as a result, our approach is both *efficient* and *compact*.

Compactness, is maybe the most striking feature: from the definition of languages and regular expressions to the correctness proof of bisimilarity, our development takes less than 1200 lines. A *self contained* (not minimal) snapshot of the library can be found at `http:\\www.cs.unibo.it\~asperti\re.tar`, and it takes about 3400 lines. The development described in [18] has about the same size, but in this case the comparison is not fair, since they only check *correctness*, but do not address neither *termination* nor *completeness*. Especially, termination for Brzozowski's procedure is a delicate issue, taking quite an effort to [13].

The formalization in [13] is unexpectedly verbose: 7414 lines, *not including* relevant fragments of the standard library. This is particularly surprising since it has been written in ssreflect [15], that is reputed to be a compact dialect of Coq. We should observe that [13] contains two bisimilarity algorithms: one slow and naif (similar to that described in [18]) and one more complex, but more difficult to prove correct (taking, respectively, 1109 and 2576 lines). The point is that the efficiency of Brzozowski's procedure largely relies on the quotient made over derivatives: associative and commutative rewriting is enough for termination, but more complex rewritings are required to get a really performant implementation.

In spite of this huge effort, the actual performance of the bisimilarity test in [13] remains modest. Let us consider a couple of examples. The first one is an encoding of Bezout's identity discussed in [13]; exploiting the fact that set inclusion can be reduced to equality expressing $A \subseteq B$ as $A \cup B = B$, the arithmetical statement

$$\forall n \geq c. \exists x, y. n = xa + yb$$

can be expressed as the following regular expression problem

$$A(a,b,c) = (0^c)0^* + (0^a + 0^b)^* \simeq (0^a + 0^b)^*$$

The second problem, borrowed from [3], consists in proving the following equality:

$$B(n) = (\epsilon + a + aa + \cdots + a^{n-1})(a^n)^* \simeq a^*$$

problem	answer	*pres*	[13]
$A(3,5,8)$	yes	0.19	2.09
$A(4,5,11)$	no	0.18	5.26
$A(4,5,12)$	yes	0.24	5.26
$A(5,6,19)$	no	0.30	31.22
$A(5,6,20)$	yes	0.43	31.23
$A(5,7,23)$	no	0.38	70.09
$A(5,7,24)$	yes	0.57	70.19

problem	answer	*pres*	[13]
$B(6)$	yes	0.15	0.29
$B(8)$	yes	0.20	1.24
$B(10)$	yes	0.26	3.98
$B(12)$	yes	0.31	10.71
$B(14)$	yes	0.45	25.04
$B(16)$	yes	0.61	53.15
$B(18)$	yes	0.80	104.16

Fig. 4. Performance

In Figure 4 we compare our technique (pres) with that of [13]; execution times are expressed in seconds and have been computed on a machine with a Pentium M Processor 750 1.86GHz and 1GB of RAM.

The main achievement of our work, is however the very notion of *pointed regular expression.* The important facts are that

1. pointed expressions are in bijective correspondence with states of DFA
2. each pointed expression has a clear and intuitive semantics
3. the relation between a state and its sons is immediate and very natural

This allows a *direct*, *intuitive* and *locally verifiable* construction of the deterministic automaton for e, that is not only convenient for formalization, but also for didactic purposes. Since their discovery, we systematically used pointed expressions for teaching the argument to students and, according to our experience, they are *largely* superior to any other method we are aware of. In our opinion, pointed regular expressions are a nice example of the kind of results we may expect from the revisitation of methods and notions of computer science and mathematics induced by mechanical formalization, and which is probably the most ambitious and challenging objective of this discipline (see e.g. [4,5]).

Acknowledgments. We thank Claudio Sacerdoti Coen for contributing to the notion of pointed regular expression and Enrico Tassi for collaborating to an earlier, partial implementation in Matita. The work has been partially supported by the European Project CerCo.[1]

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Pearson Education Inc. (2006)
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)

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

3. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science 155, 291–319 (1996)
4. Asperti, A., Armentano, C.: A page in number theory. Journal of Formalized Reasoning 1, 1–23 (2008)
5. Asperti, A., Avigad, J.: Zen and the art of formalization. Mathematical Structures in Computer Science 21(4), 679–682 (2011)
6. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita Interactive Theorem Prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)
7. 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)
8. Asperti, A., Tassi, E., Coen, C.S.: Regular expressions, au point. eprint arXiv:1010.2604 (2010)
9. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theor. Comput. Sci. 48(3), 117–126 (1986)
10. 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)
11. Brüggemann-Klein, A.: Regular expressions into finite automata. Theor. Comput. Sci. 120(2), 197–213 (1993)
12. Chang, C.-H., Paige, R.: From Regular Expressions to Dfa's using Compressed Nfa's. In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 90–110. Springer, Heidelberg (1992)
13. 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)
14. Fischer, S., Huch, F., Wilke, T.: A play on regular expressions: functional pearl. In: Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, pp. 357–368. ACM (2010)
15. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in COQ. Journal of Formalized Reasoning 3(2), 95–152 (2010)
16. Huet, G.P.: Residual theory in lambda-calculus: A formal development. J. Funct. Program. 4(3), 371–394 (1994)
17. Ilie, L., Yu, S.: Follow automata. Inf. Comput. 186(1), 140–162 (2003)
18. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. Journal of Automated Reasoning (2011) (published online)
19. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IEEE Transactions on Electronic Computers 9(1), 39–47 (1960)
20. Owens, S., Reppy, J.H., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. 19(2), 173–190 (2009)
21. Thompson, K.: Regular expression search algorithm. Communications of ACM 11, 419–422 (1968)
22. Wu, C., Zhang, X., Urban, C.: A Formalisation of the Myhill-Nerode Theorem Based on Regular Expressions (Proof Pearl). In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 341–356. Springer, Heidelberg (2011)

Using Locales to Define a Rely-Guarantee Temporal Logic

William Mansky and Elsa L. Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign,
Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302
{mansky1,egunter}@illinois.edu

Abstract. In this paper, we present an agent-based logic called Rely-Guarantee Temporal Logic (RGTL), developed using the Isabelle theorem prover. RGTL provides a formalism for expressing complex temporal-logic specifications of multi-agent systems, as well as a compositional method of reasoning about the dependencies between components in such a system. Taking advantage of Isabelle's locale functionality, we are able to express various choices about the notion of "strategy" used in the logic (e.g., memoryless/memory-based) as parameters to the semantics, whereas previously these choices were considered to define semantics for distinct variants of agent-based logics. We can then state and formally verify various aspects of RGTL, including its reasoning principles and its expressiveness relative to Alternating-time Temporal Logic (ATL), independently of the type of underlying strategies, by using locales to axiomatize the necessary requirements on strategies.

Keywords: logics for agency, temporal logic, reasoning about strategies, modular specification, Isabelle proof assistant.

1 Introduction

Alternating-time temporal logic (ATL) [2] is an extension of temporal logic to a system with multiple *players*, agents whose choices influence the evolution of a system. By introducing quantification over the *strategies* defining the future actions of some set of players, ATL provides a mechanism for formulating properties of the form "A can guarantee φ" or "A must allow φ". However, when dealing with systems containing multiple components working in concert, "can guarantee" and "must allow" are of less interest than "will guarantee" or "does not guarantee". These properties can be expressed, for instance, using the ATL-STIT language proposed by Broersen et al. [6], where STIT is an acronym for "sees to it that". Rely-Guarantee Temporal Logic (RGTL) expands on this approach, providing a formalism for expressing temporal-logic properties of and dependencies between components, as well as generalizing the notion of "strategy" from a deterministic function on states and agents to a range of potentially nondeterministic, progressively refined objects. RGTL is a logic designed to support the concepts of rely-guarantee reasoning and agency as first-class concepts, providing a flexible logic for specifying and checking requirements on complex multi-component systems.

The design of the semantics of RGTL was done using the Isabelle proof assistant, and in particular takes advantage of Isabelle's *locale* facility, which provides a mechanism for collecting and manipulating the assumptions required by various theories [3]. Through successive layers of locales, we build up the necessary framework for defining RGTL, including the underlying automata (concurrent game structures), a fundamental notion of strategies, and various axioms and operations on strategies. By minimizing the assumptions made in any given locale, we can give a general statement of the logic, independent of various distinctions that in past work have been considered to define different logics. For example, ATL with irrevocable strategies [1] has been defined in two variants, IATL (in which strategies are memoryless) and MIATL (in which strategies have unbounded memory). By abstracting away from the details of strategy computations, we are able to give a single definition of RGTL for both memoryless and memory-based strategies (as well as various other potential distinctions), which can be specialized to either case by plugging in the corresponding sublocale. Using the same approach in our analysis of expressiveness, we are able to prove that RGTL is more expressive than ATL* regardless of the type of strategies used, as long as the type of strategies is consistent across the two logics.

2 Example: A First Look

To understand the extra flexibility afforded to us by RGTL over ATL, let us consider a simple example with two agents, A and B, and a system. The system offers to each agent a toggle, which, at each instant, the agent associated with the toggle may either push or leave alone. The toggles jointly control whether a light is on or off. If, in a given instant, just one agent pushes their toggle, the light will change state: if it was on it will go off, and if it was off it will go on. If both agents either leave their toggles alone, or simultaneously push them, the light will not change state: if it was on, it will stay on, and if it was off, it will stay off.

Now let us consider the property P that at some point the light will be on and remain on from that point forward. In Linear Temporal Logic (LTL) [14], this can be stated as $\Diamond \Box \mathsf{light\_on}$, i.e., "eventually always the light is on". Obviously, if the two agents are free at each instant to choose whether to push the toggle or not, the system will display some traces that satisfy this property, but also many that do not. If we want to know whether the two players can collaborate to assure P, then we are effectively asking if there exists a trace satisfying P, which is a property that can be expressed in branching-time temporal logics such as CTL* [4]. However, if we wish to focus on what one agent can control without joint collaboration with the other, we are unable to prove any meaningful results. In particular, speaking in ATL terms, it should be clear that a single agent cannot guarantee P, and indeed must allow $\neg P$ (written as $[\![A]\!]\Box\Diamond\neg\mathsf{light\_on}$). No matter what strategy agent A pursues, there is a way for agent B to mess things up.

However, were agent A able to make use of certain properties of the behavior of agent B, then it might be possible for A to craft a strategy to always guarantee P, even if A did not know exactly what B would do at any given instant. For example, if B could definitely be relied upon to eventually stop toggling, then the strategy for A to always push the toggle on when the light is off will guarantee that eventually the light will be on and stay on. It is this kind of conditional component-wise reasoning that we aim to express and support in RGTL.

3 RGTL Syntax

Intuitively, the ATL path quantifier $\langle\langle A\rangle\rangle$ allows us to express "can-guarantee" properties; $\langle\langle A\rangle\rangle\varphi$ holds of a system when there is *some* strategy for A that ensures φ (despite the actions of the remaining agents). The dual operator, $[\![A]\!]\varphi \equiv \neg\langle\langle A\rangle\rangle\neg\varphi$, holds when for *any* strategy for A, the remaining agents can ensure φ; this can be intuitively understood as a "must-allow" property. In the case in which we have an existing strategy on which we want to check properties, neither of these operators provides the correct formalism.

Instead, we would like to say that a program *does* satisfy a property, and more generally that agent a satisfies some property P_a as long as agent b satisfies its own property P_b, which may be thought of as the *protection envelope* for agent b. The concept of the protection envelope appears in the work of Gunter et al. [9]. While it may be possible to show that a particular workflow for b satisfies a desired property, minor variations in the workflow for b may violate the property. The protection envelope is a more general property that may be satisfied by variations on b's expected workflow, while providing enough information to ensure safety of the overall system. The $\overset{A}{\Rrightarrow}$ operator is designed to facilitate this style of system specification: the left-hand side of the implication is the protection envelope for A, and the right-hand side is the property enabled by this envelope. Because of its similarity to the rely-guarantee approach originally proposed by Jones [10], we refer to this operator as the "rely-guarantee arrow".

Following CTL* and ATL*, an RGTL formula is either a *state formula* or a *path formula*; the semantics of a state formula depends only on information about the current state, while the semantics of a path formula includes assertions on possible future states. The path and state formulae of RGTL defined as:

$$\varphi ::= \varphi \wedge \varphi \mid \neg\varphi \mid \circ\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \psi$$

$$\psi ::= \mathsf{true} \mid \pi \mid \psi \wedge \psi \mid \neg\psi \mid \psi \overset{A}{\Rrightarrow} \psi \mid \Lambda\varphi$$

where $\pi \in \Pi$ is an atomic proposition and $A \subseteq \mathcal{A}$ is a set of agents. As in LTL, $\circ\varphi$ (read "next φ") asserts that φ holds in the next state along a path, while $\varphi_1\,\mathcal{U}\,\varphi_2$ (read "φ_1 until φ_2") asserts that φ_1 holds at every point along the path until the first point at which φ_2 holds. RGTL also includes the rely-guarantee operator $\overset{A}{\Rrightarrow}$, and the Λ operator, which quantifies over the outcomes of a strategy. The semantics of these operators is given in the following section.

4 Semantics

4.1 Concurrent Game Structures and Strategies

The semantics of RGTL in Isabelle is built up through a series of nested locales. Each locale introduces a set of objects and axioms defining one of the concepts needed to give semantics to RGTL formulae, and provides useful constructs and lemmas for working with that concept. Ideally, each locale introduces exactly the assumptions needed for the definitions and proofs it provides. Through this approach, our semantics remains agnostic of the underlying implementation of various features of the semantics, in particular that of the type of *strategies* introduced below.

The first locale, `CGS`, introduces the concept of a concurrent game structure, a type of automata that moves from state to state according to the actions of a set of agents. The semantics of ATL and related logics, including RGTL, are evaluated with concurrent game structures as their underlying automata.

Locale Definition 1. *A concurrent game structure (abbreviated CGS) is a tuple $(\mathcal{A}, Q, \Pi, \pi, \Sigma, e, \delta)$, where $\mathcal{A}$ is a finite and non-empty set of agents (also called players), Q is a finite set of states, Π is a finite set of atomic propositions, π is a labeling function from each state $q \in Q$ to the set of atomic propositions that hold in q, Σ is a finite set of actions available to the agents, $e : \mathcal{A} \times Q \to 2^{\Sigma}$ is a function that gives the (non-empty) set of enabled actions for each combination of agent and state, and $\delta : Q \times \Sigma^{\mathcal{A}} \to Q$ defines the transitions between states based on the actions of each agent.*

As in other agent-based logics, satisfaction of an RGTL formula is defined in terms of *strategies* for sets of agents and the *outcomes* of those strategies. Various definitions of strategies have been presented for ATL; for instance, strategies may have no memory, bounded memory, or unbounded memory [5], and may be deterministic or nondeterministic [17]. In the strategy locale for RGTL, `CGS_strategies`, we give an extremely general definition of strategies, and require only the operations needed to define the semantics of strategy-based logics.

Locale Definition 2. *Let $(\mathcal{A}, Q, \Pi, \pi, \Sigma, e, \delta)$ be a CGS.*

Let R be the type of state information*, which supports the operations* $\mathsf{current\_state}(\rho)$*,* $\mathsf{init}(q)$ *(creation of initial state information), and $\rho \cdot q$ (update with a new state).*

A strategy *is an object supporting the function $[\![\_]\!] : \mathcal{A} \times R \to 2^{\Sigma}$ such that for all agents a and state information ρ, $[\![S]\!](a, \rho)$ is a non-empty subset of $e(a, \mathsf{current\_state}(\rho))$.*

Intuitively, $[\![S]\!](a, \rho)$ is the set of actions allowed by S for agent a, given knowledge ρ of past states. In a concrete instance, ρ may be a state, finite history, or infinite history; for example, we can obtain memoryless strategies by taking $R = Q$ and letting $\mathsf{current\_state}(\mathsf{q}) = q$, $\mathsf{init}(q) = q$, and $q \cdot q' = q'$.

Given these axioms, we can define the following constructs on strategies.

Definition 1. *A strategy S is* deterministic *if for each agent a, either $|[\![S]\!](a,\rho)| = 1$ for all ρ or else $[\![S]\!](a,\rho) = e(a, \mathsf{current\_state}(\rho))$ for all ρ.*

A deterministic strategy is one that either completely determines the actions of an agent, or else places no restrictions on it. This is the type of strategies used in the original definition of ATL [2]; in general, RGTL strategies may offer any number of choices for each agent.

Definition 2. *The outcomes* out *of a strategy S and state information ρ are defined as $\mathsf{out}(S,\rho) = \{\lambda.\ \lambda_0 = \mathsf{current\_state}(\rho) \wedge \forall i.\ \exists\overline{\sigma}.\ (\forall a.\ \sigma_a \in [\![S]\!](a, \rho\cdot\lambda_{[1,i]})) \wedge \lambda_{i+1} = \delta(\lambda_i, \overline{\sigma})\}$, where $\overline{\sigma}$ is a vector of actions, one for each agent. We write λ_i for the i^{th} element of the sequence λ, $\lambda_{[i,j]}$ for the subsequence of λ starting at the i^{th} element and ending at the j^{th} element (or the empty sequence when $j < i$), and $\rho\cdot\lambda$ for ρ updated with the elements of λ.*

An infinite path λ through the underlying CGS is an *outcome* of a strategy S given state information ρ if λ starts in the current state $\mathsf{current\_state}(\rho)$ and there is a way to proceed from each state in λ to the next that is allowed by S. Note that in the case where S is deterministic and ρ is a single state q, this corresponds exactly to the original ATL definition of outcomes. As outcomes are infinite sequences of states, we make use of the theory of infinite lists from the Archive of Formal Proofs [15] to help us reason about them in Isabelle.

Definition 3. *We say that a strategy T is a* refinement *of a strategy S, written $T \sqsubseteq S$, when $[\![T]\!](a,\rho) \subseteq [\![S]\!](a,\rho)$ for each agent a and state information ρ.*

We also refer to a strategy T such that $T \sqsubseteq S$ as a *sub-strategy* of S. As one might expect, reducing the nondeterminism of a strategy reduces the set of outcomes; this result follows directly from the relevant definitions.

Lemma 1. *If $T \sqsubseteq S$, then $\mathsf{out}(T,\rho) \subseteq \mathsf{out}(S,\rho)$ for any ρ.*

As part of the `CGS_strategies` locale, we also assume several methods of deriving strategies from existing strategies, forming an implementation-agnostic algebra of strategies. The first such axiom allows us to derive strategies that ignore or presume particular state information. This will be of particular use in showing the relationship between RGTL and ATL (Section 6.3).

Locale Axiom 1. *In `CGS_strategies`, for any strategy S and state information ρ, there is a strategy T such that $\forall a\ \lambda.\ [\![S]\!](a, \rho\cdot\lambda) = [\![T]\!](a, \mathsf{init}(\mathsf{current\_state}(\rho))\cdot\lambda)$, and a strategy R such that $\forall a\ \lambda.\ [\![S]\!](a, \mathsf{init}(\mathsf{current\_state}(\rho))\cdot\lambda) = [\![T]\!](a, \rho\cdot\lambda)$.*

Second, we assume that given a potentially nondeterministic strategy with a range of possible outcomes, we can pick out a sub-strategy that produces any particular outcome.

Locale Axiom 2. *In `CGS_strategies`, for any outcome $\lambda \in \mathsf{out}(S,\rho)$, there is a strategy T such that $T \sqsubseteq S$ and $\mathsf{out}(T,\rho) = \{\lambda\}$.*

4.2 Strategies in RGTL

While these definitions are sufficient to allow us to talk about strategies and satisfaction for ATL and its variants, RGTL requires several additional strategy operators. We axiomatize these operators in the `RGTL_semantics` locale.

Locale Definition 3. *$\top$ is a strategy such that for any agent a and state information ρ, $[\![\top]\!](a, \rho) = e(a, \mathsf{current\_state}(\rho))$.*

Given a strategy S for the system and a set of agents $A \subseteq \mathcal{A}$, we define the restriction *of S to A by*

$$[\![S|_A]\!](a, \rho) = \begin{cases} [\![S]\!](a, \rho) & a \in A \\ e(a, \mathsf{current\_state}(\rho)) & a \notin A \end{cases}$$

We can use restriction to talk about strategies for individual agents or groups of agents, which place no restrictions on the behavior of the rest of the system. Note that $\top$ has the same semantics as $S|_\emptyset$ for any S.

In order to determine satisfaction of the rely-guarantee operator, we also need a mechanism for combining multiple strategies.

Locale Definition 4. *We say that two strategies S and T are* consistent *if for all input we have $[\![S]\!](a, \rho) \cap [\![T]\!](a, \rho) \neq \emptyset$. We define the* join *of two consistent strategies, written $S \sqcap T$, by $[\![S \sqcap T]\!](a, \rho) = [\![S]\!](a, \rho) \cap [\![T]\!](a, \rho)$.*

In other words, $S \sqcap T$ allows only actions that are allowed by both S and T. When S and T are inconsistent, the output of $S \sqcap T$ is ill-defined, since all strategies must allow at least one action for any input. We take care to ensure that this case does not arise in the evaluation of the satisfaction of RGTL formulae. We also assume that the $\sqcap$ operator is associative, commutative, idempotent, and has $\top$ as an identity; in other words, $\sqcap$ induces a semilattice on strategies, with $\top$ as the top element.

The $\sqcap$ operator can be shown to have the following properties with respect to refinement:

Lemma 2. *If $T \sqsubseteq S$, then $T \sqcap S' \sqsubseteq S \sqcap S'$ for any S' consistent with T.*

Lemma 3. *$(S \sqcap T)|_A \sqsubseteq S|_A$ and $(S \sqcap T)|_A \sqsubseteq T|_A$ for any consistent S and T.*

These properties are useful in establishing a framework for component-wise reasoning in RGTL (see Section 6).

4.3 RGTL Semantics

The satisfaction of a RGTL state formula is defined with respect to a CGS C, a strategy S, and state information ρ, as follows:

- $C, S, \rho \models \mathsf{true}$
- $C, S, \rho \models p$ iff $p \in \pi(\mathsf{current\_state}(\rho))$ where $p \in \Pi$ is an atomic proposition
- $C, S, \rho \models \psi_1 \wedge \psi_2$ iff $C, S, \rho \models \psi_1$ and $C, S, \rho \models \psi_2$

- $C, S, \rho \models \neg\psi$ iff $C, S, \rho \not\models \psi$
- $C, S, \rho \models \psi_1 \overset{A}{\Rrightarrow} \psi_2$ iff $\forall T.\ T|_A \sqsubseteq S|_A \wedge (\forall R.\ C, T|_A \sqcap R|_{\overline{A}}, \rho \models \psi_1) \Rightarrow$ $C, S \sqcap T|_A, \rho \models \psi_2$
- $C, S, \rho \models \Lambda\varphi$ iff $\forall \lambda \in \mathsf{out}(C, S, \rho).\ C, S, \rho, \lambda \models \varphi$

Of particular note is the semantics of the rely-guarantee arrow, which formally expresses the core of the rely-guarantee reasoning principle: $\psi_1 \overset{A}{\Rrightarrow} \psi_2$ holds when, for any strategy $T|_A$ for A that guarantees ψ_1 regardless of the behavior of the rest of the agents ($R|_{\overline{A}}$), that strategy combined with the current strategy S guarantees ψ_2. In other words, as long as $T|_A$ can be relied on to provide ψ_1, S and $T|_A$ together guarantee ψ_2.

The satisfaction of a path formula also depends on a future path λ, which in general is provided by the strategy S through evaluation of the Λ operator.

- $C, S, \rho, \lambda \models \varphi_1 \wedge \varphi_2$ iff $C, S, \rho, \lambda \models \varphi_1$ and $C, S, \rho, \lambda \models \varphi_2$
- $C, S, \rho, \lambda \models \neg\varphi$ iff $C, S, \rho, \lambda \not\models \varphi$
- $C, S, \rho, \lambda \models \circ\varphi$ iff $C, S, \rho \cdot \lambda_1, \lambda_{[1,\infty)} \models \varphi$
- $C, S, \rho, \lambda \models \varphi_1\, \mathcal{U}\, \varphi_2$ iff $\exists i.\ C, S, \rho \cdot \lambda_{[1,i]}, \lambda_{[i,\infty)} \models \varphi_2\ \wedge$ $\forall j < i.\ C, S, \rho \cdot \lambda_{[1,j]}, \lambda_{[j,\infty)} \models \varphi_1$
- $C, S, \rho, \lambda \models \psi$ iff $C, S, \rho \models \psi$

This satisfaction relation is defined as a primitive recursive function in the `RGTL_semantics` locale, and forms the basis for all of the following theorems and proofs.

5 Example: Verifying Rely-Guarantee Properties

Recall the simple light-switch system of Section 2. In this section, we will use RGTL to formally state and verify the rely-guarantee property described previously.

We begin by describing the concurrent game structure C that we will use to model the system. We have two agents, A and B, each in charge of a toggle. We will model the state of our system with a collection of boolean variables: $\mathsf{light\_on}$, which is true when the light is on in the current state; pushed_A, which is true when A's toggle was pushed in the previous state; and pushed_B, which is true when B's toggle was pushed in the previous state. Thus, our system has a total of eight states. Our variables will also act as our atomic propositions: each holds on a state exactly if it is true in the state. The actions available to each agent are either to push their toggle, or to do nothing (a τ action). In every state, both actions are enabled for each agent. Then our transition function can be described as:

$$\delta(q, (\sigma_A, \sigma_B)) = \left\{ \begin{array}{ll} \mathsf{light\_on} & = \text{if } \sigma_A = \sigma_B \text{ then } \mathsf{light\_on}\ q \text{ else } \neg\mathsf{light\_on}\ q \\ \mathsf{pushed}_A & = (\sigma_A = \mathsf{push}) \\ \mathsf{pushed}_B & = (\sigma_B = \mathsf{push}) \end{array} \right\}$$

For this example, we will take our state information to be histories, that is, finite sequences of states already seen. We can take the set of strategies to be all functions mapping elements of $\{A, B\}$ and sequences of states to non-empty subsets of $\{\tau, \mathsf{push}\}$. The strategy $\top$ is the function that assigns to each agent the full set $\{\tau, \mathsf{push}\}$ in each state. The join of two strategies is the component-wise intersection of the outputs of the strategies. We will show in Section 6.4 that the axioms of the `RGTL_semantics` locale are satisfied by this model.

Finally, recall the system property we wish to verify: that eventually the light will always be on. As in LTL, this property may be expressed in RGTL as $A\Diamond\Box\mathsf{light\_on}$ (where $\Diamond$ ("eventually") and $\Box$ ("always") can be defined in terms of the $\mathcal{U}$ operator). As stated earlier, this is not a property that one agent alone can guarantee. However, if we assume that agent B will eventually stop pushing their toggle ($A\Diamond\Box\neg\mathsf{pushed}_B$), then there is a strategy for A to pursue, namely:

$$S(a, \{\mathsf{light\_on},\ \mathsf{pushed}_A,\ \mathsf{pushed}_B\} = \left\{\begin{array}{ll} \{\tau\} & \text{if } a = A\ \wedge\ \mathsf{light\_on} \\ \{\mathsf{push}\} & \text{if } a = A\ \wedge\ \neg\mathsf{light\_on} \\ \{\tau, \mathsf{push}\} & \text{if } a = B \end{array}\right\}$$

S is a strategy for A in the sense that $S|_A = S$, that is, S only restricts the behavior of A.

Using these pieces, we can prove the following for any history ρ:

Lemma 4. $C, S, \rho \models (A\Diamond\Box\neg\mathsf{pushed}_B) \overset{B}{\Rrightarrow} (A\Diamond\Box\mathsf{light\_on})$.

Proof. By the semantics of RGTL, we can prove this by fixing a strategy T such that $T|_B \sqsubseteq S_B$, assuming that $\forall R.\ C, T|_B \sqcap R|_A, \rho \models A\Diamond\Box\neg\mathsf{pushed}_B$, and showing that $C, S \sqcap T|_B, \rho \models A\Diamond\Box\mathsf{light\_on}$. In particular, since $S|_A = S$, we may assume that $C, T|_B \sqcap S, \rho \models A\Diamond\Box\neg\mathsf{pushed}_B$. By the semantics of A, this means that along every outcome $\lambda \in \mathsf{out}(T|_B \sqcap S, \rho)$, there is some point i such that for any $j \geq i$, $\neg\mathsf{pushed}_B\ \lambda_j$. Given our labeling, this is true only if $\sigma_B = \tau$ from point i onwards, that is, if B only performs τ after point i. Now, either $\mathsf{light\_on}\ \lambda_i$, or $\neg\mathsf{light\_on}\ \lambda_i$. In the former case, the action prescribed by S for A is τ, and since B must also perform τ, the light will remain on indefinitely. In the latter case, the action prescribed by S for A is push, and since B must perform τ, the light will go on in the next state. In either case, by the above logic, once the light is on it will remain on indefinitely, as both agents continue to perform τ. Thus $C, S \sqcap T|_B, \rho \models A\Diamond\Box\mathsf{light\_on}$, and the proof is complete. □

In this manner, we can use RGTL to state and verify properties on a single agent given some assumptions on the remainder of the system. In Section 6.2, we will state a theorem that allows us to compose properties of this form to construct general specifications for a larger system.

6 Logical Properties of RGTL

Here we present various theorems that facilitate reasoning about specifications in RGTL, all of which have been formally proved in Isabelle. All theorems are proved in the context of the `RGTL_semantics` locale, and so can be generalized to any interpretation of concurrent game structures, strategies, and strategy operators.

6.1 Properties of the $\Rrightarrow$ Operator

First, we examine the behavior of the rely-guarantee operator $\overset{A}{\Rrightarrow}$ at the extremes, that is, when A is either the full set of agents $\mathcal{A}$ or the empty set.

Lemma 5. *$C, S, \rho \models \psi_1 \overset{\mathcal{A}}{\Rrightarrow} \psi_2$ iff $C, T, \rho \models \psi_1$ implies $C, T, \rho \models \psi_2$ for all $T \sqsubseteq S$.*

In other words, $\overset{\mathcal{A}}{\Rrightarrow}$ is a stronger form of implication that holds not only for the current strategy S but for all sub-strategies of S as well.

Lemma 6. *$C, S, \rho \models \psi_1 \overset{\emptyset}{\Rrightarrow} \psi_2$ iff $C, S, \rho \models \psi_2$ only if $C, T, \rho \models \psi_1$ for all T.*

This lemma shows that $\psi_1 \overset{\emptyset}{\Rrightarrow} \psi_2$ states the rather unintuitive property that ψ_2 holds under the current strategy only if ψ_1 is true under *any* strategy, i.e., ψ_1 is a constant that holds regardless of strategy. While at first this property may seem too restrictive to be of use, we can in fact use it to construct several defined operators that provide general quantification over strategies.

Definition 4. *Let $\mathsf{exS}\ \psi \equiv (\neg\psi) \overset{\emptyset}{\Rrightarrow} \mathsf{false}$ and $\mathsf{allS}\ \psi \equiv \neg\mathsf{exS}\ \neg\psi$.*

Lemma 7. *$C, S, \rho \models \mathsf{exS}\ \psi$ iff $\exists S'.\ C, S', \rho \models \psi$.*

Lemma 8. *$C, S, \rho \models \mathsf{allS}\ \psi$ iff $\forall S'.\ C, S', \rho \models \psi$.*

These operations help us bridge the gap between RGTL, in which established strategies are carried throughout a formula, and ATL, in which strategies are reselected at each strategy quantifier.

6.2 Reasoning in RGTL

The core of component-wise reasoning in RGTL is the following theorem, modeled on the rule given by Xu et al. for parallel composition in concurrent programs [16].

Theorem 1. *Suppose we have a CGS C, a strategy S, and disjoint sets of agents A and B such that $C, S|_A, \rho \models rely_A \overset{\overline{A}}{\Rrightarrow} guar_A$ and $C, S|_B, \rho \models rely_B \overset{\overline{B}}{\Rrightarrow} guar_B$. Furthermore, suppose that for all T, $C, T, \rho \models (rely_A \wedge guar_A) \Rightarrow rely_B$ and $C, T, \rho \models (rely_B \wedge guar_B) \Rightarrow rely_A$. Then $C, S|_{A \cup B}, \rho \models rely_A \overset{\overline{A \cup B}}{\Rrightarrow} guar_A \wedge guar_B$.*

Proof. We show that $C, S|_{A \cup B}, \rho \models \mathit{rely}_A \overset{\overline{A \cup B}}{\Rrightarrow} \mathit{guar}_A \wedge \mathit{guar}_B$ by fixing a strategy $T|_{\overline{A \cup B}}$ for agents not in $A \cup B$, assuming that $T|_{\overline{A \cup B}}$ guarantees rely_A for any behavior of A and B, and showing that therefore $U = S|_{A \cup B} \sqcap T|_{\overline{A \cup B}}$ satisfies $\mathit{guar}_A \wedge \mathit{guar}_B$. In particular, we may assume that $T|_{\overline{A \cup B}} \sqcap S|_B$ guarantees rely_A. Then since $C, S|_A, \rho \models \mathit{rely}_A \overset{\overline{A}}{\Rrightarrow} \mathit{guar}_A$, we know that $S|_A \sqcap T|_{\overline{A \cup B}} \sqcap S|_B = U$ guarantees guar_A.

Similarly, we may assume that $T|_{\overline{A \cup B}}$ guarantees rely_A, and thus show that $S|_A \sqcap T|_{\overline{A \cup B}}$ guarantees guar_A. Using our assumption once more, we have that $S|_A \sqcap T|_{\overline{A \cup B}}$ also satisfies rely_A, and so satisfies rely_B. Then, since $C, S|_B, \rho \models \mathit{rely}_B \overset{\overline{B}}{\Rrightarrow} \mathit{guar}_B$, we can conclude that $S|_B \sqcap S|_A \sqcap T|_{\overline{A \cup B}} = U$ satisfies guar_B as well, and the proof is complete. □

This theorem connects RGTL to the method of rely-guarantee reasoning for which it is named [10]. The pre- and post-conditions used by Xu et al. are absent, since RGTL deals with properties on infinite executions rather than terminating processes, but otherwise the rely-guarantee method of reasoning fits neatly with the $\overset{A}{\Rrightarrow}$ operator, justifying our intuitive understanding of it as the "rely-guarantee arrow". While the language of Xu et al. uses an interleaved model of concurrency, the CGS model provides true synchronization, so the disjunctive requirements on the rely- and guarantee-formulae can be replaced with stronger conjunctive conditions. Using this rule, if we prove that each component of a system satisfies its specification (its guarantee) given the protection envelope of the rest of the system, we can then conclude that the combined system satisfies the combination of each component specification.

6.3 Expressiveness

With the help of the exS operator defined in Section 6.1, we can construct an embedding of ATL* in RGTL. In particular, we can define a syntactic transformation h from a formula in ATL* to an RGTL formula as follows:

- $h_{state}(p) = p$ where $p \in \Pi$ is an atomic proposition
- $h_{state}(\neg\psi)_{state} = \neg h_{state}(\psi)$, and $h_{state}(\psi_1 \wedge \psi_2) = h_{state}(\psi_1) \wedge h_{state}(\psi_2)$
- $h_{state}(\langle\langle A \rangle\rangle \varphi) = \mathsf{exS}\ \neg(\Lambda(h_{path}(\varphi)) \overset{A}{\Rrightarrow} \mathsf{false})$
- $h_{state}(\neg\varphi)_{path} = \neg h_{path}(\varphi)$, and $h_{path}(\varphi_1 \wedge \varphi_2) = h_{path}(\varphi_1) \wedge h_{path}(\varphi_2)$
- $h_{path}(\psi) = h_{state}(\psi)$ where ψ is a state formula
- $h_{path}(\bigcirc\varphi) = \bigcirc h_{path}(\varphi)$
- $h_{path}(\varphi_1\, \mathcal{U}\, \varphi_2) = h_{path}(\varphi_1)\, \mathcal{U}\, h_{path}(\varphi_2)$

In order to show that this translation preserves the semantics of ATL*, we first must address two major disparities between RGTL and ATL. The first is revocability of strategies: while in ATL all strategies are cleared from the context at each quantification operator, RGTL may in general retain its strategies indefinitely once chosen. The use of the exS operator allows us to simulate the revocable behavior of ATL:

Lemma 9. *For any strategy S, $C, S, \rho \models h_{state}(\psi)$ iff $C, \top, \rho \models h_{state}(\psi)$, and $C, S, \rho, \lambda \models h_{path}(\varphi)$ iff $C, \top, \rho, \lambda \models h_{path}(\varphi)$.*

The second point of disparity is in the treatment of state information. In ATL, the information available to a strategy begins at the point the strategy is chosen; while it may build up knowledge of the past over its lifetime, it has no access to the states visited before reaching the strategy quantifier where it was invoked. In RGTL, by contrast, a strategy may have available the entire history built up over the course of evaluation of a formula. This gap is bridged by use of Locale Axiom 1 from Section 4; given that there exists a strategy that produces certain outcomes given some state information, we can provide one that produces the same outcomes given only the current state, and vice versa. (Note that, since the type of state information is a parameter to the `CGS_strategies` locale, the state information used may not necessarily be a history; in the case where the type of state information is instantiated to be simply the current state, the following lemma is trivial.)

Lemma 10. *For any strategy S, $C, S, \rho \models h_{state}(\psi)$ iff the single-state state information $C, S, \mathsf{init}(\mathsf{current\_state}(\rho)) \models h_{state}(\psi)$, and $C, S, \rho, \lambda \models h_{path}(\varphi)$ iff $C, S, \mathsf{init}(\mathsf{current\_state}(\rho)), \lambda \models h_{path}(\varphi)$.*

With these two differences reconciled, we can then prove the following theorem.

Theorem 2. *For any ATL* state formula ψ, path formula φ, and state information ρ, $C, \rho \models_{\mathrm{ATL}} \psi$ if and only if $C, \top, \rho \models_{\mathrm{RGTL}} h_{state}(\psi)$, and $C, \lambda \models_{\mathrm{ATL}} \varphi$ if and only if $C, \top, \mathsf{init}(\lambda_0), \lambda \models_{\mathrm{RGTL}} h_{path}(\varphi)$.*

Proof. By simultaneous induction on the structure of ψ and φ.

While most cases of the translation are straightforward, the translation of the strategy quantifier $\langle\langle A \rangle\rangle$ is of particular interest. To understand the correctness of the embedding, we must unfold the semantics of our translation for $\langle\langle A \rangle\rangle\psi$. By Lemma 7, $\mathsf{exS}\ \neg(\Lambda(\varphi) \overset{A}{\Rrightarrow} \mathsf{false})$ is true given state information ρ iff $\exists S.\ C, S, \rho \models \neg(\Lambda(\varphi) \overset{A}{\Rrightarrow} \mathsf{false})$. In general, for any formula ψ, we have that $C, S, \rho \models \neg(\psi \overset{A}{\Rrightarrow} \mathsf{false})$ iff $\exists T.\ T|_A \sqsubseteq S|_A \wedge (\forall R.\ T|_A \sqcap R|_{\overline{A}}, \rho \models \psi)$. Choosing S to be equal to T, we then have that $C, S, \rho \models \mathsf{exS}\ \neg(\Lambda(\varphi) \overset{A}{\Rrightarrow} \mathsf{false})$ iff $\exists T.\ \forall R.\ C, T|_A \sqcap R|_{\overline{A}}, \rho \models \Lambda(\varphi)$, that is, there is some strategy T for A such that for all strategies R for the remaining agents, in all outcomes, φ holds. This is precisely the definition of the strategy quantification operator $\langle\langle A \rangle\rangle$. $\square$

Thus, h is a semantics-preserving embedding of ATL* in RGTL, and we can conclude that RGTL is at least as expressive as ATL*. This proof is completed in the `RGTL_semantics` locale, and so is independent of implementation details, and in particular of the type of strategies used. In other words, we have shown that RGTL is at least as expressive as ATL* for all variants of strategies – whether memory-based, memoryless, deterministic, or nondeterministic – as long as RGTL and ATL* use the same type of strategies.

As Alur et al. have shown that model-checking for ATL* is 2EXPTIME-complete (for memory-based deterministic strategies), model-checking for RGTL with these strategies is at least 2EXPTIME-complete. Model-checking for memoryless strategies may be more tractable, since the space of memoryless strategies is sharply constrained by the size of the CGS; on the other hand, model-checking for fully nondeterministic strategies is likely to be more complex.

6.4 Concrete Interpretation

Thus far, all our reasoning has taken place inside the `RGTL_semantics` locale, under the assumption of a type of strategies supporting the various operations used in the semantics of RGTL. In order to show that these properties hold for any actual logical system, we must show that there exists an *interpretation* of the locale, i.e., a concrete instantiation of the various required types and sets that satisfies the locale's axioms. In this section, we present a model of nondeterministic strategies with unbounded memory, and use it to construct an interpretation of the `RGTL_semantics` locale.

Definition 5. *A* nondeterministic strategy with unbounded memory *on a CGS* $(\mathcal{A}, Q, \Pi, \pi, \Sigma, e, \delta)$ *is a function* $S : \mathcal{A} \times Q^+ \rightarrow 2^\Sigma$ *such that for any agent* a *and history* ρ*,* $S(a, \rho) \subseteq e(a, \mathsf{last}(\rho))$ *and* $S(a, \rho)$ *is non-empty.*

Using this definition, we can construct an interpretation of `RGTL_semantics` in two steps. More precisely, we will construct a proof that the `CGS` locale, extended with this notion of strategies, is a sublocale of `RGTL_semantics`; that is, we will provide concrete interpretations for strategies and strategy operators, but continue to axiomatize the definition of a concurrent game structure. Since `RGTL_semantics` is built on top of the `CGS_strategies` locale, we begin by showing that `CGS` extended with this notion of strategies is a sublocale of `CGS_strategies`.

Lemma 11. *A* `CGS` *along with its nondeterministic strategies with unbounded memory is an instance of* `CGS_strategies`*.*

Proof. To prove this, we must show that the type of state information supports the operations $\mathsf{current\_state}(\rho)$, $\mathsf{init}(\rho)$, and $\rho \cdot q$, and that the type of strategies supports the operation $[\![S]\!]$. Our type of state information is Q^+, the set of finite non-empty sequences of states in Q, and so we can define $\mathsf{current\_state}(\rho) = \mathsf{last}(\rho)$, $\mathsf{init}(q) = q$, and $\rho \cdot q = \rho \cdot q$ in the sense of concatenation of sequences. Similarly, our strategies are already functions from $\mathcal{A} \times Q^+$ to 2^Σ that are non-empty and consistent with e, so we may define $[\![\_]\!]$ to be simply the identity function.

In addition, we must show that the two strategy-creation axioms are satisfied by our interpretation. Given a strategy S with certain behavior on a sequence ρ, the strategy $\lambda a\ \rho'.\ S(a, \rho_{[0,|\rho|-1)} \cdot \rho')$ produces the same behavior on $\mathsf{current\_state}(\rho)$; similarly, for a given sequence ρ, if S has some behavior on $\mathsf{current\_state}(\rho)$, the strategy $\lambda a\ \rho'.\ S(a, \rho'_{[|\rho|-1,|\rho'|)})$ has the same behavior on ρ. Finally, we must show that for any outcome $\lambda \in \mathsf{out}(S, \rho)$, there exists a sub-strategy $T \sqsubseteq S$ such that $\mathsf{out}(T, \rho) = \{\lambda\}$. This is the most complex part of the proof of interpretation; putting aside the technical details, the intuition is to provide the strategy that, for each history of the form $\rho \cdot \lambda_{[1,i]}$, produces a vector of actions $\overline{\sigma}$ such that $\delta(\lambda_i, \overline{\sigma}) = \lambda_{i+1}$. We know that such a vector exists at each point i because S has produced λ as an outcome through precisely such a vector, and so can create a strategy that at each step mirrors the behavior of S in producing λ. □

Since our strategy interpretation function is the identity function, it follows naturally that the strategy operators are given by their axiomatized semantics.

Definition 6. *Let* $\top = \lambda a\ \rho.\ e(a, \mathsf{current\_state}(\rho))$, $S|_A = \lambda a\ \rho.$ if $a \in A$ then $S(a, \rho)$ else $e(a, \mathsf{current\_state}(\rho))$, *and* $S \sqcap T = \lambda a\ \rho.\ S(a, \rho) \cap T(a, \rho)$.

Lemma 12. *Under these definitions, a `CGS` with nondeterministic strategies with unbounded memory is an instance of `RGTL_semantics`.*

Proof. We must simply show that each strategy operator satisfies its axioms, which follows directly from the definitions of the operators. □

7 Conclusion

In this paper, we have presented the rely-guarantee-based temporal logic RGTL, and demonstrated its use as a compositional method for verifying properties of multi-agent concurrent systems. We have shown a semantics-preserving embedding of ATL* into RGTL parameterized by the type of strategies used by the agents, so that we can be assured of the relative expressiveness of RGTL as long as certain assumptions hold on the type of strategies. We also have presented an instantiation of the generic type of strategies, and demonstrated that one common notion of strategy satisfies the necessary assumptions. All theorems have been formally verified in the Isabelle theorem prover, giving us a strong assurance of their correctness.

While we believe RGTL has appeal as a logic in its own right, it has also benefited considerably from the use of Isabelle in its development. By building RGTL on top of Isabelle's locale system, we are able to define several variants of the logic – deterministic, nondeterministic, memoryless, memory-based – with a single semantic function. The use of locales on the one hand allows us to state our theorems and write our proofs in their full generality, and on the other hand forces us to explicitly state our assumptions and demonstrate that they are satisfied by the intended models. The building blocks of our proofs, including the locales `CGS` and `CGS_strategies`, may be reused in the Isabelle development of other strategy-based logics, allowing for strategy-agnostic expressiveness results such as ours with respect to ATL*. The strategy operators defined in our locales may also have uses beyond the semantics of RGTL; for instance, our join operation $\sqcap$ on strategies is related to the $\dagger$ operation used in IATL to update a CGS with a strategy [1]. Recent research in agent-based formalisms has given rise to a plethora of ATL-related logics (see for instance Brihaye et al.'s taxonomy of ATL variants [5]); we believe that movement towards a general, strategy-agnostic framework for defining the semantics of these logics will considerably simplify the process of formally stating, verifying, and comparing them.

The Isabelle development described in this paper can be found online at `https://netfiles.uiuc.edu/mansky1/www`.

8 Future Work

While RGTL represents a step forward in expressing properties of multi-agent systems, there are still various features of real-world programs that are not reflected in the logic. For instance, using the ordinary temporal logic connectives of LTL/CTL/ATL, it is difficult to compare values across states in a path; a property such as "the value of x always increases" is non-intuitive to state and prove. The Temporal Logic of Actions (TLA) [11] is a variant of LTL that addresses this problem by expanding the atomic propositions to relations over pairs of states ("actions"). Work is in progress to extend RGTL to the setting of actions, allowing a more concise intuitive description of software- and workflow-like properties.

One clear area for further work is the problem of *incomplete information*; in practice, not every player in a game may have full access to the current state. There has been extensive work on this problem as it relates to ATL and similar logics; see for instance the work of Dima et al. [8] Through approaches such as imposing equivalence classes on histories, thus limiting each player's knowledge of the environment, we can more accurately model partial-knowledge scenarios, for instance parallel programs in which each thread can only access certain variables.

Strategy Logic [7] is a logic related to ATL, with facilities for more general quantification over strategies. We are currently exploring an extension of strategy logic with strategy satisfaction and refinement, which we believe to be strictly more expressive than RGTL. Since strategy logic is known to be decidable, this may provide a method of proving the decidability of model-checking for RGTL.

While the complexity of model-checking full RGTL is as yet undetermined, in practice, the full expressiveness of RGTL may not be required. For instance, note that in the case study, arbitrary nesting of the $\overset{A}{\Rrightarrow}$ operator is not required to express the protection-envelope properties. If we restrict our language to a Horn-clause-like fragment of RGTL, in which the rely-guarantee operator is used in a strictly "positive" manner, the model-checking problem may become more tractable.

9 Related Work

In previous work by Nieto [13], the original Owicki-Gries method (as refined by Jones [10]) was formalized in Isabelle/HOL. This approach relies on axiomatic semantics (Hoare-style reasoning) rather than temporal logic for program verification, but provides a similar principle of modular reasoning.

Our notion of strategy satisfaction is similar to and borrows concepts from the STIT-extension of ATL proposed by Broersen et al. [6]; however, the STIT extension is restricted to the case of deterministic strategies, and does not address the subtleties of strategy combination and refinement. We also build on the work of Yasmeen on varieties of ATL and logics with strategies [17].

Mogavero et al. [12] define a semantics for ATL* that, like RGTL, retains knowledge of the execution complete history ρ rather than only the current state and future execution. In Mogavero's "relentful" approach, the temporal operators are evaluated on the combination of history and future execution, so that for instance $\diamond\varphi$ is satisfied if φ held sometime in the past. In our semantics, players may take history into account when making their strategic decisions, but each temporal formula is still evaluated beginning at the moment its strategy comes into effect, with the history serving only to increase the range of possible strategies.

GL (game logic) is a generalization of ATL/ATL* proposed by Alur et al. [2], which allows arbitrary quantification over sets of strategies. In combination with strategy satisfaction and refinement, this provides a mechanism similar to (but not equivalent to) the rely-guarantee operator of RGTL. Strategy logic, mentioned above, can also be seen as an extension of GL with more flexible strategy quantification.

ATL with Strategy Contexts and Bounded Memory [5] is another step towards greater control over the strategy quantification of ATL, providing several quantification operators that allow switching between revocable (ATL-style) and irrevocable (IATL-style) use of strategies. We have not yet determined the expressiveness of RGTL with respect to ATL$^*_{sc}$, which would be an interesting area for future work.

Acknowledgements. This material is based upon work supported in part by NASA Contract NNA10DE79C and NSF Grant 0917218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA or NSF.

References

1. Ågotnes, T., Goranko, V., Jamroga, W.: Alternating-time temporal logics with irrevocable strategies. In: Proceedings of the 11th Conference on Theoretical Aspects of Rationality and Knowledge, TARK 2007, pp. 15–24. ACM, New York (2007), http://doi.acm.org/10.1145/1324249.1324256
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
3. Ballarin, C.: Locales and Locale Expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 34–50. Springer, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-24849-1_3
4. Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. In: POPL 1981: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 164–176. ACM, New York (1981)

5. Brihaye, T., Da Costa, A., Laroussinie, F., Markey, N.: ATL with Strategy Contexts and Bounded Memory. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 92–106. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-92687-0_7
6. Broersen, J., Herzig, A., Troquard, N.: A STIT-Extension of ATL. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 69–81. Springer, Heidelberg (2006)
7. Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. Inf. Comput. 208, 677–693 (2010), http://dx.doi.org/10.1016/j.ic.2009.07.004
8. Dima, C., Enea, C., Guelev, D.P.: Model-checking an alternating-time temporal logic with knowledge, imperfect information, perfect recall and communicating coalitions. In: GANDALF, pp. 103–117 (2010)
9. Gunter, E.L., Yasmeen, A., Gunter, C.A., Nguyen, A.: Specifying and analyzing workflows for automated identification and data capture. In: HICSS, pp. 1–11 (2009)
10. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5, 596–619 (1983), http://doi.acm.org/10.1145/69575.69577
11. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16, 872–923 (1994), http://doi.acm.org/10.1145/177492.177726
12. Mogavero, F., Murano, A., Vardi, M.Y.: Relentful Strategic Reasoning in Alternating-Time Temporal Logic. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 371–386. Springer, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939141.1939162
13. Prensa Nieto, L.: The Rely-Guarantee Method in Isabelle/HOL. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 348–362. Springer, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1765712.1765738
14. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS 1977, pp. 46–57. IEEE Computer Society, Washington, DC (1977), http://dx.doi.org/10.1109/SFCS.1977.32
15. Trachtenherz, D.: Infinite Lists. Archive of Formal Proofs 2011 (2011)
16. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing 9, 149–174 (1997)
17. Yasmeen, A.: Formalizing operator task analysis. Ph.D. thesis, University of Illinois at Urbana-Champaign, USA (2011)

Charge!
A Framework for Higher-Order Separation Logic in Coq

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal

IT University of Copenhagen

Abstract. We present a comprehensive set of tactics for working with a shallow embedding of a higher-order separation logic for a subset of Java in Coq. The tactics make it possible to reason at a level of abstraction similar to pen-and-paper separation-logic proof outlines. In particular, the tactics allow the user to reason in the embedded logic rather than in the concrete model, where the stacks and heaps are exposed. The development is generic in the choice of heap model, and most of the development is also independent of the choice of programming language.

1 Introduction

Higher-order separation logic [3] (HOSL) is an extension of separation logic that allows for quantification over predicates in both the assertion logic (the logic of pre and postconditions) and the specification logic (the logic of Hoare triples). Higher-order separation logic has proved useful for modular reasoning about programs that use shared mutable data structures and data abstraction, via quantification over resource invariants, and for reasoning about various forms of higher-order programming (higher-order functions, code pointers, interfaces in object-oriented programming) [4,10,12,15,8,2].

This paper describes Charge!, a separation-logic verification tool that aims to (1) prove full functional correctness of Java-like programs using higher-order separation logic, (2) produce machine-checkable correctness proofs, (3) work as close as possible to how informal separation logic proofs are carried out on pen and paper, and (4) automate tedious first-order reasoning where possible.

The first and second goal virtually mandate that we build the tool inside an existing proof assistant for higher-order logic such as Coq. All other tools with these properties that we know of [6,16,11,1] have been built this way; building them from the ground up instead would be a very ambitious undertaking.

To achieve the third goal, it is important that the user has the feeling of reasoning in the program logic, rather than in the model of the logic. For separation logic, this entails reasoning about the linear fragments of the logic, like on paper, rather than reasoning about concrete machine states, such as the stack or the heap, and disjointness properties of various heap fragments. In our own earlier work [2], based on a shallow embedding of higher-order separation logic in the Coq proof assistant, this goal was achieved and the programs were verified without the user ever seeing an explicit heap or a stack in the proof context. However, as the built-in tactics of Coq are not designed to handle the linear fragments of separation logic, our proofs were long, tedious, and to a large degree focusing mainly on rewriting the proof state modulo associativity and commutativity of separating conjunction.

In this article, we improve on that work and present a comprehensive set of tactics for working with a shallow embedding of a higher-order separation logic for a subset of Java. The tactics make it possible to reason at a level of abstraction similar to pen and paper proofs. In particular, the tactics allow the user to reason in the embedded logic rather than in the concrete model, where the stacks and heaps are exposed. The use of these tactics significantly reduce the size of the proofs compared to our earlier development [2].

The remainder of this paper is structured as follows. In Section 2, we discuss how we deal with program variables in Charge! – how the relation between logical variables and program variables is made as transparent as possible in a proof assistant. In Section 3, we present an extended example of how a user can prove correctness of an implementation of binary search trees using Charge!. Then, in Section 4 we present the most prominent tactics in our development and discuss how they affect the goal states in Coq. Sections 5 covers specifics on how the tactics are implemented and Section 6 touches on how Charge! can be adapted to new languages. Section 7 covers related work, and Section 8 concludes.

This paper is structured top-down: we only include the definitions that are required for the exposition and focus on describing what the tactics do before going into how they do it. The interested reader can find all of the core definitions of the assertion and specification logics as well as examples on how these are used to verify object-oriented programs in [2]. Our Coq development can be downloaded from http://itu.dk/research/charge.

2 Program Variables

A key to approaching pen-and-paper style of reasoning is to handle program variables naturally. We feel that the variables-as-resource approach [13] is too unnatural for this purpose. The approach of McCreight [11], while practical, also looks very different from pen-and-paper proofs. Our philosophy is that a user must be able to blur the boundaries between logical variables and program variables, as is often done on paper, in a proof assistant. In this section we discuss how we make this blurring formal.

It is typical in a richly typed separation logic to distinguish between program variables, whose values are restricted to the types offered by the programming language, and logical variables, which can be lists, functions, and other types offered by the logic. Program variables occurring in triples $\{P\}c\{Q\}$ refer to local (stack) variables in c. For example, the heap-write rule can be written

$$\frac{}{\{x.f \mapsto e\}\ x.f := e'\ \{x.f \mapsto e'\}}$$

Here, $x : \mathit{var}$ is a program variable name, typically a pointer to an object, $e, e' : \mathit{expr}$ are programming language expressions, and $f : \mathit{field}$ is a field name.

The expression $x.f \dot{\mapsto} e$ reads that the expression e can be found at the memory address $x.f$. There are free program variables in these assertions since x is itself a program variable, and e and e' may contain program variables.

In pen-and-paper theories of Hoare logic, it is often imagined that the fragment of mathematics required for the proof, such as the theory of lists, is recreated inside the assertion logic in a version where program variables may occur in assertions. When encoding this in a proof assistant, it is not enough to imagine it, and actually doing it would be far too much work. We want to reuse existing theories as they are, and we want to build new theories without being concerned about program variables before there even is a program.

This includes the theory of heap assertions, which is independent of program variables [17,11] even though the two are very often defined together and become inseparable [1,8]. One primitive in this theory is the points-to predicate $(\mapsto) : \mathit{val} \to \mathit{field} \to \mathit{val} \to (\mathit{heap} \to \mathit{Prop})$[1], where *val* is the type of data values for the programming language under consideration. To use the points-to predicate in pre and postconditions, like we saw in the heap-write rule above, we *lift* it to $(\dot{\mapsto}) : \mathit{expr} \to \mathit{open}\ \mathit{field} \to \mathit{expr} \to \mathit{open}\ (\mathit{heap} \to \mathit{Prop})$.

An *open* T is intuitively a T that may have free program variables:

$$\mathit{open}\ T \triangleq \mathit{stack} \to T \qquad \mathit{stack} \triangleq \mathit{var} \to \mathit{val} \qquad \mathit{expr} \triangleq \mathit{open}\ \mathit{val}$$

In general, the operator lift_n will lift constants and functions of type $(T_1 \to \cdots \to T_n \to U)$ into $(\mathit{open}\ T_1 \to \cdots \to \mathit{open}\ T_n \to \mathit{open}\ U)$. Unfolding the definition of the lifted points-to predicate $(\dot{\mapsto})$ makes the write-rule read

$$\frac{}{\{(\mathit{lift}_3\ (\mapsto))\ (\mathit{ve}\ x)\ (\mathit{lift}_0\ f)\ e\}\ x.f := e'\ \{(\mathit{lift}_3\ (\mapsto))\ (\mathit{ve}\ x)\ (\mathit{lift}_0\ f)\ e'\}}$$

where *ve* is the injection from variable names to expressions. Further unfolding the definitions of lift_3 and *ve*, the rule reads

$$\frac{}{\{\lambda s.\ (s\ x).f \mapsto (e\ s)\}\ x.f := e'\ \{\lambda s.\ (s\ x).f \mapsto (e'\ s)\}}$$

Note that this is only an exposition – the user will never see an explicit stack in the proof context. We see that nothing very deep is involved in this treatment of program variables, but it offers some very convenient properties. First, it avoids an explicit mentioning of stacks s. Second, the property of substitutions that $(e_1.f \dot{\mapsto} e_2)\{^e/_x\} = e_1\{^e/_x\}.f \dot{\mapsto} e_2\{^e/_x\}$ follows from the definition of the lifting and is independent of the definition of $\mapsto$. As long as all operators in an assertion are lifted, substitutions will propagate automatically over the connectives and be applied when they reach the program variables – Coq does this computationally, hence it is very fast, and the tactics do not have to reason about any meta-theoretical properties of substitution.

[1] For the sake of exposition, we assume that our heap assertions are of type $\mathit{heap} \to \mathit{Prop}$. See [2] for the full definition.

3 Example

To introduce our tactics, we use a library of binary search trees. The specification and the code in this example are transliterations of the sorted_bintree example that comes with the VeriFast tool [9]. By stepping through a method of the library command by command, we demonstrate how our tactics modify the goal of the proof assistant while allowing the user to reason strictly at the level of the assertion logic – in this section there are no explicit stacks, there are no explicit heaps, and there are no explicit substitutions. They are all present in the background, but they are hidden from the user.

The library has methods init(x) for creating a singleton tree, contains(t, x) for membership query and add(t, x) for adding a single element. We specify the methods in terms of the representation predicate $tree(t, b)$, which describes the memory footprint of a tree with root pointer t and contents b, with $b : bintree$ defined as $b \triangleq empty \mid node\ n\ b\ b$, where n is an integer. We also create a predicate *indorder* of type $bintree \to Prop$ that holds if b is a proper search tree, and a function *t_cont* of type $bintree \to \mathbb{Z} \to bool$ that assumes that b is a search three and checks for membership of a value in the standard way. We also have a *TreeRec*-predicate that describes the footprint of a binary search tree

$$\begin{array}{l}
TreeRec\ t\ b \triangleq \\
\quad \mathsf{match}\ b\ \mathsf{with} \\
\qquad \mid empty \qquad\quad \Rightarrow t = null \\
\qquad \mid node\ v\ bl\ br \Rightarrow t.\mathsf{value} \mapsto v * \exists l.\ \exists r.\ t.\mathsf{left} \mapsto l * t.\mathsf{right} \mapsto r * \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad TreeRec\ l\ bl * TreeRec\ r\ br \\
\quad \mathsf{end}
\end{array}$$

where t is a pointer to the heap, and b is of type *bintree*. The connective $*$ is the standard separating conjunction where $p * q$ reads that p is true for one part of the heap, and q is true for another disjoint part of the heap. Finally, we use this definition to define $tree\ t\ b \triangleq TreeRec\ t\ b \wedge inorder\ b$. The library is specified as follows, where $\dot{f}\ x_1\ \cdots\ x_n$ means $(lift_n\ f)\ x_1\ \cdots\ x_n$, and $\mathsf{C}{::}\mathsf{m}(\vec{\mathsf{x}}) \mapsto \{p\}\_\{q\}$ means that the class C contains a method m, with arguments $\vec{\mathsf{x}}$, that has a specification with the precondition p and the postcondition q.

$$\begin{array}{l}
Tree\_spec \triangleq \forall b. \\
\qquad\qquad \mathsf{Tree::init(x)} \mapsto \{\dot{\top}\}\_\{\dot{tree}\ \mathsf{ret}\ (\dot{node}\ \mathsf{x}\ \dot{empty}\ \dot{empty})\} \wedge \\
\qquad\qquad \mathsf{Tree::contains(t,x)} \mapsto \{\dot{tree}(\mathsf{t}, \dot{b})\}\_\{\dot{tree}(\mathsf{t}, \dot{b}) \dot{\wedge} \mathsf{ret} \doteq t\_\dot{cont}(\dot{b}, \mathsf{x})\} \wedge \\
\qquad\qquad \mathsf{Tree::add(t,x)} \mapsto \begin{array}{l} \{\dot{tree}(\mathsf{t}, \dot{b}) \dot{\wedge} b \neq \dot{empty} \dot{\wedge} t\_\dot{cont}(\dot{b}, \mathsf{x}) \doteq \dot{false}\}\_ \\ \{\dot{tree}(\mathsf{t}, tree\_\dot{add}(\dot{b}, \mathsf{x}))\} \end{array}
\end{array}$$

This is a predicate in our specification logic; for a full disclosure, see [2]. It lists the methods of a specification as well as their pre and their postconditions. The program variable ret in the postconditions store the return value of the method. In this example, we focus on verifying the contains-method. We need the following lemmas

$$\begin{aligned}\mathsf{TreeRec\_null} &\triangleq \forall b.\ (\mathit{TreeRec\ null}\ b \vdash \mathit{TreeRec\ null}\ b \wedge b = \mathit{empty})\\ \mathsf{TreeRec\_not\_null} &\triangleq \forall t\ b.\ t \neq \mathit{null} \rightarrow (\mathit{TreeRec}\ t\ b \vdash \mathit{TreeRec}\ t\ b \wedge b \neq \mathit{empty})\end{aligned}$$

Both lemmas follow directly from the definition of *TreeRec*. The connectives in the predicates are not lifted. The contains-method is defined as follows.

```
contains(t, x) =
  if t $\doteq$ null then ret := $\mathit{false}$ else
    v := t.value; if x $\doteq$ v then ret := $\mathit{true}$
                  else if x $\dot{<}$ v then l := t.left; ret := t.contains(l, x)
                  else r := t.right; ret := t.contains(r, x)
```

The operators in the conditional statements are lifted from Coq's standard library. To demonstrate how our tactics operate on this method, we step through the proof one command at a time. After some initial boiler-plate setup, our state looks as follows. For the exposition, we omit lifting constants such as $\dot{b}$.

$$\frac{b : \mathit{bintree}}{\{\dot{\mathit{tree}}\ \mathtt{t}\ b\}\mathbf{if}\ \mathtt{t} \doteq \mathtt{null}\ \mathbf{then}\ \mathtt{ret} := \mathit{false}\ \mathbf{else}\ \ldots\{\dot{\mathit{tree}}\ \mathtt{t}\ b\ \dot{\wedge}\ \mathtt{ret} \doteq \dot{\mathit{t\_cont}}\ b\ \mathtt{x}\}}$$

We denote the postcondition of this triple with $\mathcal{P}$. Using the forward-tactic generates two sub-goals – one for each branch of the if-statement.

$$1\ \frac{b : \mathit{bintree}}{\left\{\begin{matrix}\dot{\mathit{tree}}\ \mathtt{t}\ b\ \dot{\wedge}\\ \mathtt{t} \doteq \mathit{null}\end{matrix}\right\}\mathtt{ret} := \mathit{false}\{\mathcal{P}\}} \qquad 2\ \frac{b : \mathit{bintree}}{\left\{\begin{matrix}\dot{\mathit{tree}}\ \mathtt{t}\ b\ \dot{\wedge}\\ \mathtt{t} \dot{\neq} \mathit{null}\end{matrix}\right\}\mathtt{v} := \mathtt{t.value};\ \ldots\{\mathcal{P}\}}$$

We start by proving subgoal 1. There is only one command, and applying the forward-tactic provides the following proof obligation.

$$\frac{b : \mathit{bintree} \qquad n : \mathbb{Z} \qquad H : \mathit{inorder}\ b}{\mathit{TreeRec}\ \mathtt{null}\ b \vdash \mathit{false} = \mathit{t\_cont}\ b\ n}$$

A few things have happened here. First of all, since the only command in the triple has been evaluated, the user is left with an assertion logic entailment to prove, the stack has been applied, and all liftings have been evaluated. Secondly, the program variable x has been replaced with an integer n, representing its value on the stack. Thirdly, the equivalence $\mathtt{t} = \mathit{null}$ has been applied. Finally, the *tree*-predicate has been evaluated and its non-spatial components (that do not depend on the heap) have been placed in the context. To prove the goal, we must infer that b is empty. The command `sl_apply TreeRec_null` applies the lemma *TreeRec_null* using forward-reasoning and places the non-spatial parts of the consequent in the context.

$$\frac{b : \mathit{bintree} \qquad n : \mathbb{Z} \qquad H : \mathit{inorder}\ b \qquad H_1 : b = \mathit{empty}}{\mathit{TreeRec}\ \mathtt{null}\ b \vdash \mathit{false} = \mathit{t\_cont}\ b\ n}$$

The command `sl_auto` then solves the goal. We now proceed to prove goal 2. We cannot immediately apply the forward-tactic as the precondition does not assert what t.value is on the heap. The command `unfold tree; triple_nf` unfolds the *tree*-predicate and places the non-spatial predicate *inorder b* in the context.

$$\frac{b : \mathit{bintree} \qquad H : \mathit{inorder}\ b}{\{\dot{\mathit{TreeRec}}\ \mathtt{t}\ b\ \dot{\wedge}\ \mathtt{t} \dot{\neq} \mathit{null}\}\mathtt{v} := \mathtt{t.value};\ \ldots\{\mathcal{P}\}}$$

More precisely, `triple_nf` places the goal in a normal form; this is discussed in more detail in sections 4 and 5.3. The command `sl_apply TreeRec_not_null` applies the lemma *TreeRec_not_null* in a forward-reasoning style to the precondition of the triple, again moving the non-spatial components of the consequent of the lemma to the context. Remember that the lemma is defined as an assertion logic formula, and its connectives are not lifted.

$$\frac{b : \mathit{bintree} \qquad H : \mathit{inorder}\ b \qquad H_1 : b \neq \mathit{empty}}{\{\dot{\mathit{TreeRec}}\ \mathtt{t}\ b\ \dot{\wedge}\ \mathtt{t} \dot{\neq} \mathit{null}\}\mathtt{v} := \mathtt{t.value};\ \ldots\{\mathcal{P}\}}$$

The command `destruct b; [congruence| clear` H_1`]` does case-analysis on b. The proof for the case where b is empty is trivial as there is a contradiction in H_1. For the remaining case, H_1 is not needed and is cleared from the context.

$$\frac{v : \mathbb{Z} \qquad b_1 : \mathit{bintree} \qquad b_2 : \mathit{bintree} \qquad H : \mathit{inorder}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)}{\left\{\begin{array}{l}\dot{\mathit{TreeRec}}\ \mathtt{t}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)\\ \dot{\wedge}\ \mathtt{t} \dot{\neq} \mathit{null}\end{array}\right\}\begin{array}{l}\mathtt{v} := \mathtt{t.value};\\ \mathcal{B}\end{array}\left\{\begin{array}{l}\dot{\mathit{tree}}\ \mathtt{t}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)\ \dot{\wedge}\\ \mathtt{ret} \doteq \dot{\mathit{t\_cont}}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)\ \mathtt{x}\end{array}\right\}}$$

We denote the postcondition of this triple with $\mathcal{P}'$, and the rest of the program $\mathcal{B}$. The forward-tactic can now be applied as reducing *TreeRec* provides the content of t.value.

$$\frac{\begin{array}{l}v : \mathbb{Z} \qquad b_1 : \mathit{bintree} \qquad b_2 : \mathit{bintree} \qquad x_1 : \mathit{val} \qquad x_2 : \mathit{val}\\ H : \mathit{inorder}\ (\mathit{node}\ v\ b_1\ b_2)\end{array}}{\left\{\begin{array}{l}\mathtt{t.value} \dot{\mapsto} v \mathbin{\dot{*}} \mathtt{t.left} \dot{\mapsto} x_1 \mathbin{\dot{*}} \mathtt{t.right} \dot{\mapsto} x_2 \mathbin{\dot{*}} \dot{\mathit{TreeRec}}\ x_1\ b_1 \mathbin{\dot{*}}\\ \dot{\mathit{TreeRec}}\ x_2\ b_2\ \dot{\wedge}\ (\mathtt{v} \doteq v\ \dot{\wedge}\ \mathtt{t} \dot{\neq} \mathit{null})\end{array}\right\}\mathcal{B}\{\mathcal{P}'\}}$$

Here the *TreeRec*-predicate has been unfolded and its existentially quantified variables have been extracted to the context. We denote the context of this goal with $\mathcal{C}$ and the spatial component of the precondition with $\mathcal{S}$. The forward-tactic, following the structure of the code, again splits the conditional into two cases. For space reasons, we only cover the first case.

$$\frac{\mathcal{C}}{\{\mathcal{S}\ \dot{\wedge}\ (\mathtt{x} \doteq \mathtt{v}\ \dot{\wedge}\ \mathtt{v} \doteq v\ \dot{\wedge}\ \mathtt{t} \dot{\neq} \mathit{null})\}\mathtt{ret} := \mathit{true}\{\mathcal{P}'\}}$$

Since there is only one command in the triple, applying the forward-tactic leaves the user to prove the following entailment.

$$\frac{\mathcal{C} \qquad k : \mathit{val} \qquad H_2 : k \neq \mathit{null}}{\left(\begin{array}{l}k.\mathtt{value} \mapsto v * k.\mathtt{left} \mapsto x_1 * k.\mathtt{right} \mapsto x_2\\ * \mathit{TreeRec}\ x_1\ b_1 * \mathit{TreeRec}\ x_2\ b_2\end{array}\right) \vdash \left(\begin{array}{l}\mathit{tree}\ k\ (\mathit{node}\ v\ b_1\ b_2) \wedge\\ \mathit{true} = \mathit{t\_cont}\ (\mathit{node}\ v\ b_1\ b_2)\ v\end{array}\right)}$$

Here, the program variable `t` is evaluated to k, and `x` is replaced by v as they are equivalent. To prove the entailment we must prove that $t\_cont\ (node\ v\ b_1\ b_2)\ v$ holds, which it does by definition. Moreover, to prove that $tree\ k\ (node\ v\ b_1\ b_2)$ holds we must prove that $inorder\ (node\ v\ b_1\ b_2)$ holds, which we have from the context, and that $TreeRec\ k\ (node\ v\ b_1\ b_2)$ holds, which assuming that we instantiate the existential quantifiers of *TreeRec* correctly, is provable directly from the hypothesis of the entailment. The tactic `sl_simpl` solves the goal.

The rest of the proof follows the same pattern and is not more complicated. For the recursive method call, the precondition is proven as a separate assertion logic entailment, but also that follows the same pattern.

One of the main points of the trace above is to demonstrate what is not there as much as what is there. There are three notable things that are not in the trace: there are no visible stacks, there are no visible heaps, and there are no visible substitutions. They are all present, and they all play important roles, but they are never exposed to the user.

4 Tactics

For the rest of the paper, we will split assertions into three different categories: spatial assertions that depend on both the heap and the stack, denoted by t, u, or v, pure assertions that depend only on the stack, denoted by p, q, or r, and propositional assertions that depend on neither the heap nor the stack, denoted by P, Q, or R. We will denote assertions that can be either spatial, pure, or propositional with a, b, or c. Propositional assertions can be viewed as the standard *Prop*-sort in Coq. There are injections from propositional assertions to pure assertions to spatial assertions, but we leave these implicit in the presentation.

Like most custom-made tactics, we make use of existential variables in Coq. An existential variable is a variable in Coq's meta-logic. It has a type, but it has not yet been assigned a value. It can be thought of as a hole in the proof waiting to be filled. Existential variables will be preceded by a ? (for instance $?x$, $?y$, or $?z$). A valid proof can have no uninstantiated existential variables.

Tactics are split into two sub-categories – those that operate on the assertion logic, and those that operate on Hoare-triples. Both types of tactics require, and enforce, that the goal is in a normal form. Neither the premise of an entailment nor the precondition of a triple may contain existential quantifiers or propositional assertions; if they do, they are extracted to the Coq context. Moreover, in the case of triples, pure and spatial assertions are kept separate. More formally, the following goals are in normal form

$$\frac{\overrightarrow{H : P}}{\{(t_1 * \cdots * t_n) \land (p_1 \land \cdots \land p_m)\}c\{a\}} \qquad \frac{\overrightarrow{H : P}}{t_1 * \cdots * t_n \vdash a}$$

where $\overrightarrow{H : P}$ are the premises (zero or more) in the Coq-context. In both cases, t_1 to t_n and p_1 to p_m are atomic, i.e. they contain no further occurrences of $*$ and $\land$ respectively. We say that $t_1 * \cdots * t_n$ is a linear assertion. The reason that pure assertions are kept in the precondition for triples and not in entailments is that pure and propositional assertions are indistinguishable in the assertion logic – the stack has already been fully applied and all liftings have been computed. In Section 5.3 we cover how to rewrite a goal to normal form.

4.1 Tactics on the Assertion Logic

All of the tactics for the assertion logic are language and memory-model independent. We achieve this by using the notion of separation algebras by Calcagno et al. [5]. For a full exposition, see [2], but in a nutshell, as long as the user provides a memory model that satisfies the axioms of separation algebras, all of the following tactics can be applied.

sl_simpl. This tactic attempts to simplify an entailment. All modifications to the goal are safe in the sense that the tactic will not make a goal unprovable. It assumes that the goal is in normal form, and given an entailment $t \vdash a$ does the following simplifications:

– Split a into a spatial component u and a propositional component P. Split the goal and simplify $t \vdash u$ and $t \vdash P$ independently.
– For each sub-formula of u, step through t to see if it is present there as well. If so, remove it from both assertions.
– Remove every sub-formula of P that is present in the Coq-context.
– If u contains an existential quantifier, replace it with an existential variable and rerun the simplification. However, this step rolls back unless the simplifier manages to solve the assertion under the binder completely; an incorrect guess of the existential variable can otherwise lead to an unprovable goal.

The reason that the goal is split in the first step, and before the spatial components are simplified, is that the spatial components are often needed to prove propositional assertions. If the spatial simplification is done before the split, the tactic can make the goal unprovable.

The sl_simpl tactic is parametric on another tactic that guides the simplifier when instantiating existential variables. This tactic dictates what safe instantiations are, i.e., which instantiations are allowed even if the entire assertion under the quantifier cannot be discharged by the tactic. As a default, this tactic is the fail-tactic; it will never succeed, and no instantiation is considered safe. However, for our Java-fragment, we allow the simplifier to instantiate existential variables if either they are checked for equality under the binder, or if they appear in the range of a pointsto-predicate. For instance, the entailment $o.f \mapsto v \vdash \exists x\ y.\ o.f \mapsto x * i.g \mapsto w \wedge x = y$ is simplified to $\mathit{true} \vdash i.g \mapsto w$ even though the goal is not solved completely.

sl_auto. This tactic is a more aggressive version of sl_simpl. It unfolds the definition of the available representation predicates, and simplifies commonly occurring sub-expressions using rewriting tactics. Finally it runs sl_simpl. This heuristic can put the goal in an unprovable state, and the tactic will never be applied automatically by any of the other tactics.

sl_apply. Standard tactics in Coq like apply or rewrite do not work in the desired way when reasoning with entailments. In Section 3, we use lemmas TreeRec_null and TreeRec_not_null to modify the proof goal, but neither the goal nor the lemmas are in a form that apply or rewrite can use in the intended way. The tactic sl_apply is designed to allow for forward reasoning in the following manner: Assume that we have a goal with an entailment in normal form $\overrightarrow{H : P} \to t \vdash a$ and a lemma $\mathcal{L}$ that we wish to apply that has the form $\forall \vec{x}.\ S_1\ \vec{x} \to \cdots S_n\ \vec{x} \to (b\ \vec{x} \vdash c\ \vec{x})$ where the variables $\vec{x}$ can be of any Coq type. The first step of the tactic is to replace all universally-quantified variables in $\mathcal{L}$ with existential variables and to split b and c into its spatial and propositional components, leaving the lemma in the form $S_1 \to \cdots S_n \to (u \wedge Q \vdash v \wedge R)$ where $b \dashv\vdash u \wedge Q$, $c \dashv\vdash v \wedge R$, and all quantifiers $\vec{x}$ have been replaced by existential variables that are free in u, v, Q, R, and S_1 to S_n. The next step is to frame u out of t, i.e., find a t' such that $t \dashv\vdash u * t'$. If this is successful, the tactic will leave the user to prove the following goals

$$\frac{\overrightarrow{H : P} \qquad H_1 : R}{v * t' \vdash a} \qquad \frac{\overrightarrow{H : P}}{t \vdash Q} \quad \frac{\overrightarrow{H : P}}{t \vdash S_1} \cdots \frac{\overrightarrow{H : P}}{t \vdash S_n}$$

where the first goal is the result of the original goal state after the application of $\mathcal{L}$, and the rest are the proofs of the propositional premises of $\mathcal{L}$. If the tactic is unable to find t', it will fail. The existential variables that are introduced in place of the quantifiers $\vec{x}$ are typically unified by Coq when t' is obtained or when Q or S_1 to S_n are proven. Uninstantiated variables are left in the goal. This behaviour is similar to the eapply-tactic in Coq.

4.2 Tactics on Triples

Unlike entailments, the predicates in triples contain program variables. A typical triple can have the form $\{t \mathbin{\dot{*}} u \mathbin{\dot{\wedge}} p\}c\{b\}$ where $\dot{*}$ and $\dot{\wedge}$ are the lifted versions of $*$ and $\wedge$ respectively, as described in Section 2. One of the more common rules in separation logic is the rule of consequence, which allows the pre and post-conditions of a triple to be rewritten. Since triples operate on lifted assertions, and entailments operate on standard ones, our rule of consequence has a slightly different form than the standard one.

$$\frac{\forall s.\ (a\ s \vdash a'\ s) \qquad \{a'\}c\{b'\} \qquad \forall s.\ (b'\ s \vdash b\ s)}{\{a\}c\{b\}}\ \textsc{RoC}$$

By applying a stack s to the lifted assertions, we obtain standard assertions. Even though this rule exposes the stack s, it is only used in intermediate steps of the tactic and the user will never see an explicit stack.

sl_apply. We extend the sl_apply-tactic from Section 4.1 to work on triples as well as entailments. The general idea is to allow forward-reasoning by rewriting the precondition of a triple using the rule of consequence and the sl_apply-tactic for entailment. However, the tactics described so far are not sufficient. To demonstrate, we attempt to rewrite the triple $\{(a \mathrel{\dot{-\!*}} b) \mathbin{\dot{*}} a \mathbin{\dot{*}} d\}c\{e\}$ to $\{b \mathbin{\dot{*}} d\}c\{e\}$ using a modus ponens rule for $-\!*$ that states that $\forall a\ b.\ (a \mathrel{-\!*} b) * a \vdash b$. Remember that the connectives are not lifted in the lemmas we apply.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b\ s * d\ s \vdash ?x\ s}\ ???}{(a\ s \mathrel{-\!*} b\ s) * a\ s * d\ s \vdash ?x\ s}\ {-\!*}\text{–MP}}{\forall s.\ (((a \mathrel{\dot{-\!*}} b) \mathbin{\dot{*}} a \mathbin{\dot{*}} d)\ s \vdash ?x\ s)}\ \forall\text{–I} \qquad \dfrac{\vdots}{\{?x\}c\{?y\}} \qquad \dfrac{\dfrac{}{?y\ s \vdash e\ s}\ \mathit{Refl.}}{\forall s.\ (?y\ s) \vdash (e\ s)}\ \forall\text{–I}}{\{(a \mathrel{\dot{-\!*}} b) \mathbin{\dot{*}} a \mathbin{\dot{*}} d\}c\{e\}}\ \text{RoC}$$

Applying the rule of consequence generates the existential variables $?x$ and $?y$ for the pre and the postcondition respectively. Instantiating $?y$ is straightforward, and follows immediately by reflexivity of $\vdash$. Instantiating $?x$ is more problematic. First, we introduce the stack s; the stack then propagates over the lifted connectives resulting in assertions with corresponding un-lifted connectives. Coq does this automatically. We then apply the modus-ponens lemma. To conclude, we need to unify $?x$ with $b \mathbin{\dot{*}} d$, in effect reversing the computation that un-lifted the connectives. This, however, Coq is not able to do automatically – the proof does go through if the user manually instantiates $?x$ but for large proofs this quickly becomes tedious. We require a tactic that will transform the assertion $b\ s * d\ s$ to $(b \mathbin{\dot{*}} d)\ s$. How we solve this is described in Section 5.2.

forward. The forward-tactic is the work horse tactic of Charge!. Given a triple $\{p\}c_1; \cdots; c_n\{q\}$ the tactic symbolically executes the command c_1, given that its requisites are met by the precondition p, rewriting p to a new predicate p'. The user is left to prove either the triple $\{p'\}c_2; \cdots; c_n\{q\}$, or, if the triple initially had only one command, the entailment $p' \vdash q$. The tactic only works for goals in normal form.

charge. The charge-tactic is the tactic that gives our framework its name. The tactic repeatedly applies the forward-tactic until either forward fails or provides the user with more than one subgoal to prove.

5 Tactic Building Blocks

We have several automatic heuristics that solve frequently occurring sub-goals of our tactics. The tactics sl_simpl and sl_apply use a framing tactic that attempts to find one occurrence of a spatial formula in another and remove that instance; the sl_apply-tactic for triples require a tactic that lifts all connectives of an assertion; finally, most tactics require that triples and entailments are in normal form, hence we have a tactic that transforms a goal to normal form. These tactics work using a combination of hint-databases and reflective tactics.

5.1 Framing

In order to frame the spatial assertion u out of t, we have to find a t' such that $t \dashv\vdash t' * u$. This is achieved by rewriting t modulo commutativity and associativity of $*$. Since we know that t is spatial, we do not have to cover the cases of there being any pure assertions or occurrences of standard conjunction in t. The first step is to define a predicate $\mathit{Frame}\ t\ u\ t' \triangleq t \dashv\vdash t' * u$ that holds if framing u out of t results in t'. This predicate is then inserted into the proof wherever framing is required. In the derivation

$$\dfrac{\mathit{Frame}\ t\ u\ ?x \qquad \dfrac{\vdots}{?x * u \vdash a}}{t \vdash a}$$

an existential variable $?x$ is introduced, and the job of the framing tactic is to find a solution for the predicate $\mathit{Frame}\ t\ u\ ?x$, instantiating $?x$ in the process. The following inference rules accomplish this, assuming that t is linear, which the normal form guarantees.

$$\dfrac{}{\mathit{Frame}\ t\ \mathit{true}\ t} \qquad \dfrac{\mathit{Frame}\ u\ t\ u'' \qquad \mathit{Frame}\ u''\ t'\ u'}{\mathit{Frame}\ u\ (t * t')\ u'} \qquad \dfrac{t = u}{\mathit{Frame}\ (t * t')\ u\ t'}$$

$$\dfrac{\mathit{Frame}\ t'\ u\ t''}{\mathit{Frame}\ (t * t')\ u\ (t * t'')} \qquad \dfrac{t = u}{\mathit{Frame}\ t\ u\ \mathit{true}}$$

We add these rules in a left-to-right priority order to a hint database. The Coq auto-tactic is then used to solve the predicate.

5.2 Lifting Connectives

Coq will reduce any term in the form $(\dot{f}\ a_1\ \cdots\ a_n)\ s$ to $f\ (a_1\ s)\ \cdots\ (a_n\ s)$, but as is demonstrated in Section 4.2, we need a tactic to reverse this computation. Similarly to framing, we have a predicate $\mathit{Lift}\ a\ b \triangleq a = b$ that in effect is a wrapper for standard Leibniz-equality. This predicate is then inserted into the proof derivations where required. For instance, in Section 4.2 we need to lift the term $b\ s * d\ s$ to $(b\ \dot{*}\ d)\ s$ when using the sl_apply-tactic in a triple. Inserting the *Lift*-predicate in the derivation accomplishes this

$$\dfrac{\dfrac{\mathit{Lift}\ (b\ s * d\ s)\ (?x\ s)}{b\ s * d\ s \vdash ?x\ s}}{\cdots}$$

and similarly to the *Frame*-predicate, the tactics instantiate the existential variable $?x$ when proving the predicate. We add the following inference rules to a hint database in order to prove occurrences of the *Lift*-predicate.

$$\dfrac{}{\mathit{Lift}\ (s\ x)\ ((ve\ x)\ s)} \qquad \dfrac{\mathit{Lift}\ a_1\ (x_1\ s) \quad \cdots \quad \mathit{Lift}\ a_n\ (x_n\ s)}{\mathit{Lift}\ (f\ a_1\ \cdots\ a_n)\ ((\dot{f}\ x_1\ \cdots\ x_n)\ s)} \qquad \dfrac{}{\mathit{Lift}\ a\ a}$$

The first rule reverse a variable lookup on the stack; the second lifts any n-ary function; the final one is the base case that will fire when everything is lifted as far as possible. In our formalisation, we have one hint for every arity of function. We also have hints for a few other cases, like un-applied substitutions, quantifiers, and if-then-else-statements.

5.3 Normal Form

Most tactics require that the goal is in normal form. Recall that the normal form ensures that, whether the goal is a triple or an entailment, that all existential variables and propositional assertions in the precondition have been extracted to the Coq-context. We create a tactic that given an assertion a, obtains a spatial assertion t, a pure assertion p, and a propositional assertion P, such that $a \dashv\vdash \exists\vec{x}.\ (t\ \vec{x} \wedge p\ \vec{x}) \wedge P\ \vec{x}$ where neither t, p, or P, contain any existential quantifiers. When such an assertion is in the precondition of an entailment or a triple, extracting the existentially quantified variables and propositional assertions to the Coq context is straightforward. The tactic that converts an assertion to normal form is a mix of reflective tactics, and hint-databases.

A Deep Embedding of Assertions. We create a Galina term that represents an assertion in normal form. The normal form requires that all existentially quantified variables are at the top level – this means that we must reason about open terms in order to describe the spatial, pure, and propositional predicates that appear under the binders. In effect, these assertions will be n-ary functions where n is the number of free logical variables in the assertion. We parametrise the deep embedding with a list of types corresponding to the types that have been existentially quantified so far. That list of types is converted to a tuple and the n-ary assertions are converted to unary ones that take one member of this tuple type, rather than a sequence of members of each binding type.

The type $exs\ Ts$ takes a list of types Ts and returns a tupled version of that list ($exs\ [] = ()$, $exs\ [\mathbb{Z},\ val,\ bool] = (\mathbb{Z},\ (val,\ (bool,\ ())))$, et c.). An open term is a tuple of three lists: ts, of type $list\ (exs\ Ts \rightarrow (heap \rightarrow Prop))$, and ps and Ps of type $list\ (exs\ Ts \rightarrow Prop)$. Intuitively, ts is a list of spatial terms separated by the $*$-operator, and ps and Ps are lists of pure and propositional terms respectively, separated by the $\wedge$-operator. We will use the notation $\langle ts,\ ps,\ Ps\rangle$ for such a tuple and give it the type $deep\_asn\ Ts$, where Ts is the list of types that the components of the tuple are parametrised on. Note that the type of the lists for pure and propositional assertions are of the same type since we cannot distinguish between these types of assertions in the assertion logic. Finally, we create a function $[T]d$ that given a term d of type $deep\_asn\ (T :: Ts)$ closes the term and returns a term of the type $deep\_asn\ Ts$. Moreover, we will use π_1 and π_2 to denote projection of the first and the second element out of tuples respectively. To demonstrate, the deep embedding of the assertion $\exists x\ y.\ o.f\mapsto x * i.g\mapsto w \wedge x = y$ is $[val][val]\langle[\lambda t.\ o.f\mapsto(\pi_1\ t),\ \lambda t.\ i.g\mapsto w]_*,\ [],\ [\lambda t.\ \pi_1\ t = \pi_1(\pi_2\ t)]\rangle$

Transforming an Assertion to Normal Form. The transformation of assertions to normal form is done using hint databases. The first step is to create an evaluation function *eval* of type $deep\_asn\ Ts \rightarrow exs\ Ts \rightarrow (heap \rightarrow Prop)$ that given an open term in normal form and a tuple instantiating the free variables, returns an equivalent assertion. We will write $[\![d]\!]_x$ for $eval\ d\ x$. Next, we define the following assertion $NF\ d\ a\ x \triangleq [\![d]\!]_x \dashv\vdash a\ x$ that holds if $[\![d]\!]_x$ evaluates to $a\ x$, given two open assertions, one deeply embedded d and one shallowly embedded a, and a tuple x that instantiates the free variables of d and a. This assertion is then inserted into proof trees when an assertion needs to be transformed to normal form. For instance, the following derivation puts the precondition of a triple in normal form.

$$\dfrac{\dfrac{\dfrac{\dfrac{NF\ ?y\ (a\ s)\ ()}{a\ s \vdash [\![?y]\!]_{()}} \qquad \dfrac{Lift\ ([\![?y]\!]_{()})\ (?x\ s)}{[\![?y]\!]_{()} \vdash ?x\ s}}{a\ s \vdash ?x\ s}\,\textit{Trans.}}{\forall s.\ (a\ s \vdash ?x\ s)}\,\forall\text{–I} \qquad \dfrac{\vdots}{\{?x\}c\{?y\}} \qquad \dfrac{\dfrac{}{?z\ s \vdash b\ s}\,\textit{Refl.}}{\forall s.\ (?z\ s \vdash b\ s)}\,\forall\text{–I}}{\{a\}c\{b\}}\,\text{RoC}$$

The evaluation order of the predicates is important. Proving the predicate $NF\ ?y\ (a\ s)\ ()$ instantiates $?y$, obtaining an assertion in normal form. Proving the predicate $Lift\ ([\![?y]\!]_{()})\ (?x\ s)$ in turn instantiates $?x$ and lifts all connectives, allowing us to prove the triple, but with the precondition in normal form.

The next step is to create the hint-database that proves occurrences of the *NF*-predicate. We create two merge functions *merge_nf_sc* and *merge_nf_and*, both of type $deep\_asn\ Ts \rightarrow deep\_asn\ Ts \rightarrow deep\_asn\ Ts$ and written with the infix operators $\circledast$ and $\textcircled{\wedge}$ respectively. The merger functions and the evaluation function are designed such that the following inferences hold.

$$\dfrac{NF\ d_a\ a\ x \qquad NF\ d_b\ b\ x}{NF\ (d_a \circledast d_b)\ (\lambda y.\ a\ y * b\ y)\ x} \qquad \dfrac{NF\ d_a\ a\ x \qquad NF\ d_b\ b\ x}{NF\ (d_a \textcircled{\wedge} d_b)\ (\lambda y.\ a\ y \wedge b\ y)\ x}$$

$$\dfrac{\forall y : T.\ NF\ d\ (\lambda z.\ a\ (\pi_1\ z)\ (\pi_2\ x))\ (y,\ x)}{NF\ ([T]d)\ (\lambda z.\ \exists y : T.\ a\ y\ z)\ x} \qquad \dfrac{}{NF\ \langle [t],\ [],\ [] \rangle\ t\ x}$$

$$\dfrac{}{NF\ \langle [],\ [p],\ [] \rangle\ p\ x} \qquad \dfrac{}{NF\ \langle [],\ [],\ [P] \rangle\ P\ x}$$

The design of the merging functions is a bit intricate. When merging the terms d_a and d_b, their top level existential quantifiers are traversed and added in sequence to the resulting term. The difficulty comes when merging the two open terms – this requires a bit of work, and is left out for space reasons.

Recall that we cannot distinguish between pure and propositional assertions in the assertion logic, i.e. $[\![\langle [],\ [p],\ [] \rangle]\!]_x \dashv\vdash [\![\langle [],\ [],\ [p] \rangle]\!]_x$. When turning a propositional assertion to normal form, the tactics will check if there syntactically exists a stack in the assertion – if so, it is classified as pure, otherwise as propositional. It is important that this classification is correct or a pure assertion can end up in the Coq-context, rather than in the precondition of a triple.

6 Custom Hoare Triples

The core of Charge! is designed to be language independent – the majority of the tactics are usable regardless of language or memory model. Once a language is defined, and all meta-theoretical properties have been proven, adapting the language to Charge! is relatively straightforward. In this section we demonstrate how to incorporate a standard read-rule from separation logic into Charge!. In separation logic, the Hoare-triple for the read rule often has the form

$$\frac{x \neq y \qquad x \notin fv\ e}{\{y.f \mapsto e\} x := y.f \{y.f \mapsto e \wedge x = e\}}\ \textsc{Read}$$

and stores the value of the expression e found at the memory location $y.f$ in the program variable x. There is also a side condition stating that x must not be a free variable in e and disjoint from y. This rule provides a minimal footprint of the read-command, but is often not directly usable as it requires the goal to be in a very specific form.

In [2] we provided an alternative read-rule that does not require the precondition to be of a certain shape, or impose any freshness conditions on x.

$$\frac{a \vdash y.f \mapsto e}{\{a\} x := y.f \{\exists v.\ a\{^v/_x\} \wedge x = e\{^v/_x\}\}}\ \textsc{ReadEnt}$$

When this rule is used, we need to prove that $y.f \mapsto e$ can be inferred from a. This is typically done by framing $y.f \mapsto e$ out of a. The substitutions perform the alpha-renamings required to enforce the side-conditions of the Read-rule. We use the tactics from Section 5 to create a new version of the rule that assumes that the goal is in normal form before it is applied, and ensures that the goal stays in normal form.

$$\frac{\begin{array}{l}\forall s.\ \exists u\ v\ ps.\ (p\ s \to \mathit{Frame}\ (a\ s)\ ((s\ y).f \mapsto v)\ u) \wedge \\ \qquad \mathit{Lift}\ v\ (e\ s) \wedge \mathit{PureBase}\ (\lambda t.\ (p\{^{\pi_1\ t}/_x\})\ s)\ ps \wedge \\ \qquad \left[\!\!\left[\begin{array}{l}[val]\langle[(\lambda t.\ a\{^{\pi_1\ t}/_x\}\ s)]_*, \\ \qquad [(\lambda t.\ s\ x = e\{^{\pi_1\ t}/_x\}\ s) :: ps,\ []\rangle]\end{array} \right]\!\!\right]_{()} = ?b\ s\end{array}}{\{a \wedge p\} x := y.f \{?b\}}\ \textsc{ReadNF}$$

This rule is a bit intimidating, but solvable by the tactics described so far. When applied, the quantifiers are introduced and existential variables created for the existentially quantified variables. We assume that the postcondition of the triple is an existential variable. If this is not the case, the rule of consequence can be used to obtain an existential variable for the postcondition. The *Frame*-predicate corresponds to the premise of the ReadEnt-rule, and the pure facts p can be used when proving the predicate. The range of the pointsto-predicate v, which is instantiated when *Frame* is proven, is then lifted using the *Lift*-predicate instantiating the expression e in the process. The predicate *PureBase* p ps instantiates ps to the empty list if p reduces to *true*, and $[p]$ otherwise – this is to avoid cluttering up the precondition with *true*-predicates. The evaluation of the normal form evaluates to the postcondition of ReadEnt. All of our triple-rules are in a similar form.

7 Related Work

Adapting proof assistants to reason in separation logic has been proposed before. Some of the earliest work is an unpublished article by Appel [1] where he creates a family of tactics that reasons about a small imperative language. The core philosophy is the same as ours – the user should be able to reason in the separation logic, not in its model, and there should never be an explicit stack or a heap in the proof context.

In later work, McCreight expanded on Appel's ideas and created a comprehensive set of tactics for verifying Cminor programs using separation logic in Coq [11]. This seminal piece of work drastically cut down proof script sizes, yet their approach differs from ours. In McCreight's work, the user will find the heap in the proof context. Where we have an entailment of the form $a * b \vdash c * d$, McCreight unfolds the definition of entailment exposing the heap $(a * b)\ m \rightarrow (c * d)\ m$, and the antecedent of the implication is then moved to the Coq context. The reason this approach works fine is that the definition of $*$ is never unfolded and even though the heap is exposed, the user never has to reason about sub-heaps or their disjointness-properties. One of the main motivations of our work was that we wanted to see whether or not it is possible to retain the simplicity of McCreight's tactics while keeping with the overall philosophy of Appel's ideas and strictly reason inside the separation logic. We claim that we have achieved this. One point of comparison is that all three formalisations have verified the standard in-place list reversal algorithm. In [1], Appel uses 200 lines and 795 words to verify this program by count of wc; McCreight uses 68 lines and less than 400 words [11]. We use 25 lines and 105 words. These numbers do not include the definition of the program, just the proofs themselves.

Other work includes the Bedrock framework by Chlipala [6]. Similar to Charge!, Bedrock strives to automate the tedium of program verification using separation logic in Coq. The focus on Bedrock lies on low level languages, including support to work with hardware registers.

Another interactive approach is Holfoot, by Tuerk [16], which verifies Smallfoot specifications inside HOL4. This approach is similar to ours in that the core of Holfoot also builds on the theories of abstract separation algebras by Calcagno et al. [5]. Holfoot also has impressive automation results, but to the best of our knowledge does not handle object-oriented programs or nested triples [14].

Another Coq-framework for separation logic by Dockins et al. is the MSL-library [7]. It provides extensive meta-theoretical results of separation algebras of different flavours, however it currently has very few tactics.

One very prominent tool for program verification is VeriFast [9]. VeriFast allows the user to write, specify, and compile C and Java-programs and prove their correctness. The approach is mostly interactive and, as the name suggests, fast. However, the tool provides no formal proof of program correctness. The binary tree example presented in Section 3 is taken from the VeriFast web-site.

8 Conclusion

We have developed Charge! – a comprehensive framework for verifying the correctness of Java-like programs using higher-order separation logic in Coq. Our tactics allow the user to focus on the actual program verification, as opposed to manually proving all of the tedious and repetitive steps that proofs of full functional program correctness typically require. Moreover, the work-flow is very close to the style of reasoning used for pen-and-paper proof outlines in separation logic, allowing users to freely exchange logical variables and program variables in the assertion logic predicates. Charge! is memory-model independent, and the modular design of the tactics makes adding new language features and commands simple and straightforward.

References

1. Appel, A.W.: Tactics for separation logic, Draft (January 2006), `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`
2. Bengtson, J., Jensen, J.B., Sieczkowski, F., Birkedal, L.: Verifying Object-Oriented Programs with Higher-Order Separation Logic in COQ. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 22–38. Springer, Heidelberg (2011)
3. Biering, B., Birkedal, L., Torp-Smith, N.: BI Hyperdoctrines and Higher-Order Separation Logic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 233–247. Springer, Heidelberg (2005)
4. Biering, B., Birkedal, L., Torp-Smith, N.: BI-hyperdoctrines, higher-order separation logic, and abstraction. ACM Trans. Program. Lang. Syst. 29(5) (2007)
5. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proceedings of LICS, pp. 366–378 (2007)
6. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI, pp. 234–245 (2011)
7. Dockins, R., Hobor, A., Appel, A.W.: A Fresh Look at Separation Algebras and Share Accounting. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 161–177. Springer, Heidelberg (2009)
8. Dodds, M., Jagannathan, S., Parkinson, M.J.: Modular reasoning for deterministic parallelism. In: Proceedings of POPL (2011)
9. Jacobs, B., Piessens, F.: The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U. Leuven (August 2008)
10. Krishnaswami, N.R., Aldrich, J., Birkedal, L., Svendsen, K., Buisse, A.: Design patterns in separation logic. In: Proceedings of TLDI, pp. 105–116 (2009)
11. McCreight, A.: Practical Tactics for Separation Logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 343–358. Springer, Heidelberg (2009)
12. Nanevski, A., Ahmed, A., Morrisett, G., Birkedal, L.: Abstract Predicates and Mutable ADTs in Hoare Type Theory. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 189–204. Springer, Heidelberg (2007)
13. Parkinson, M.J., Bornat, R., Calcagno, C.: Variables as resource in Hoare logic. In: Proceedings of LICS, pp. 137–146. IEEE (2006)

14. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare Triples and Frame Rules for Higher-Order Store. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 440–454. Springer, Heidelberg (2009)
15. Svendsen, K., Birkedal, L., Parkinson, M.: Verifying Generics and Delegates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 175–199. Springer, Heidelberg (2010)
16. Tuerk, T.: A Formalisation of Smallfoot in HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 469–484. Springer, Heidelberg (2009)
17. Varming, C., Birkedal, L.: Higher-order separation logic in Isabelle/HOLCF. Electr. Notes Theor. Comput. Sci. 218, 371–389 (2008)

Mechanised Separation Algebra

Gerwin Klein, Rafal Kolanski, and Andrew Boyton

[1] NICTA, Sydney, Australia*
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{Gerwin.Klein,Rafal.Kolanski,Andrew.Boyton}@nicta.com.au

Abstract. We present an Isabelle/HOL library with a generic type class implementation of separation algebra, develop basic separation logic concepts on top of it, and implement generic automated tactic support that can be used directly for any instantiation of the library. We show that the library is usable by multiple example instantiations that include structures such as a heap or virtual memory, and report on our experience using it in operating systems verification.

Keywords: Isabelle, Separation Logic.

1 Introduction

The aim of this work is to support and significantly reduce the effort for future separation logic developments in Isabelle/HOL by factoring out the part of separation logic that can be treated abstractly once and for all. This includes developing typical default rule sets for reasoning as well as automated tactic support for separation logic. We show that both of these can be developed in the abstract and can be used directly for instantiations.

The library supports users by enforcing a clear axiomatic interface that defines the basic properties a separation algebra provides as the underlying structure for separation logic. While these properties may seem obvious for simple underlying structures like a classical heap, more exotic structures such as virtual memory or permissions are less straight-forward to establish. The library provides an incentive to formalise towards this interface, on the one hand forcing the user to develop an actual separation algebra with actual separation logic behaviour, and on the other hand rewarding the user by supplying a significant amount of free infrastructure and reasoning support.

Neither the idea of separation algebra nor its mechanisation is new. Separation algebra was introduced by Calcagno et al [2] whose development we follow, transforming it only slightly to make it more convenient for mechanised instantiation. Mechanisations of separation logic in various theorem provers are plentiful, we have ourselves developed multiple versions [5,6] as have many others. Similarly a

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work was in part funded by AOARD grant #FA2386-11-1-4070.

number of mechanisations of abstract separation algebra exist, e.g. by Tuerk [7] in HOL4, by Bengtson et al [1] in Coq, or by ourselves in Isabelle/HOL [5].

The existence of so many mechanisations of separation logic is the main motivation for this work. Large parts of developing a new separation logic instance consist of boilerplate definitions, deriving standard properties, and often re-invented tactic support. While separation algebra is used to justify the separation logic properties of specific developments [5], or to conduct a part of the development in the abstract before proceeding to the concrete [1,7], the number of instantiations of these abstract frameworks so far tends to be one. In short, the library potential of separation algebra has not been exploited yet in a practically re-usable way. Such lightweight library support with generic interactive separation logic proof tactics is the contribution of this paper.

A particular feature of the library presented here is that it does not come with a programming language, state space, or a definition of hoare triples. Instead it provides support for instantiating your own language to separation logic. This is important, because fixing the language, even if it is an abstract generic language, destroys most of the genericity that separation algebra can achieve. We have instantiated our framework with multiple different language formalisations, including both deep and shallow embeddings. The library is available for download from the Archive of Formal Proofs [4].

In Sec 2 we show the main interface of the separation algebra class in Isabelle/HOL and describe how it differs from Calcagno et al. Sec 3 describes the generic tactic support, and Sec 4 describes our experience with example instances.

2 Separation Algebra

This section gives a brief overview of our formulation of abstract separation algebra. The basic idea is simple: capture separation algebra as defined by Calcagno et al [2] with Isabelle/HOL type class axioms, develop separation logic concepts in the abstract as far as can be done without defining a programming language, and instantiate simply using Isabelle's type class instantiations. This leads to a lightweight formalisation that carries surprisingly far.

Calcagno et al define separation algebra as *a cancellative, partial commutative monoid* $(\Sigma, \cdot, u)$*. A partial commutative monoid is given by a partial binary operation where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined.* [2]

For a concrete instance, think of the carrier set as a heap and of the binary operation as map addition. The definition induces separateness and substate relations, and is then used to define separating conjunction, implication, etc. Since the cancellative property is needed primarily for completeness and concurrency, we leave it out at the type class level. If necessary, it could be introduced in a second class on top. The definition above translates to the following class axioms.

$$x \oplus 0 = \mathit{Some}\ x \qquad x \oplus y = y \oplus x \qquad a \mathbin{++} b \mathbin{++} c = (a \mathbin{++} b) \mathbin{++} c$$

where $\mathit{op}\ \oplus :: \text{'a} \Rightarrow \text{'a} \Rightarrow \text{'a}\ \mathit{option}$ is the partial binary operator and $\mathit{op}\ {++} :: \text{'a}\ \mathit{option} \Rightarrow \text{'a}\ \mathit{option} \Rightarrow \text{'a}\ \mathit{option}$ is the $\oplus$ operator lifted to strict partiality with None ++ x = None. From this the usual definitions of separation logic can be developed. However, as to be expected in HOL, partiality makes the $\oplus$ operator cumbersome to instantiate; especially the third axiom often leads to numerous case distinctions. Hence, we make the binary operator total, re-using standard + as syntax. Totality requires us to put explicit side conditions on the laws above and to make disjointness ## a parameter of the type class leading to further axioms. The full definition of separation algebra with a total binary operator is

```
class sep_algebra = zero + plus +
  fixes op ##:: 'a ⇒ 'a ⇒ bool
  assumes x ## 0  and  x ## y ⟹ y ## x  and  x + 0 = x
  assumes x ## y ⟹ x + y = y + x
  assumes ⟦x ## y; y ## z; x ## z⟧ ⟹ x + y + z = x + (y + z)
  assumes ⟦x ## y + z; y ## z⟧ ⟹ x ## y
  assumes ⟦x ## y + z; y ## z⟧ ⟹ x + y ## z
```

This form is precisely as strong as Calcagno et al's formulation above in the sense that either axiom set can be derived from the other. The last two axioms are encapsulated in the original associativity law. The more intuitive form $x \mathbin{\#\#} y \Longrightarrow x + y \mathbin{\#\#} z = (x \mathbin{\#\#} z \wedge y \mathbin{\#\#} z)$ is strictly stronger.

While 7 axioms may seem a higher burden than 3, the absence of lifting and type partiality made them smoother to instantiate in our experience, in essence guiding the structure of the case distinctions needed in the first formulation.

Based on this type class, the definitions of basic separation logic concepts are completely standard, as are the properties we can derive for them. Some definitions are summarised below.

$$\begin{array}{ll}
P \wedge\!* \; Q & \equiv \lambda h.\ \exists x\ y.\ x \mathbin{\#\#} y \wedge h = x + y \wedge P\ x \wedge Q\ y \\
P \longrightarrow\!* \; Q & \equiv \lambda h.\ \forall h'.\ h \mathbin{\#\#} h' \wedge P\ h' \longrightarrow Q\ (h + h') \\
x \preceq y & \equiv \exists z.\ x \mathbin{\#\#} z \wedge x + z = y \\
\square & \equiv \lambda h.\ h = 0 \\
\bigwedge\!*\ Ps & \equiv \mathit{foldl}\ (\mathit{op}\ \wedge\!*)\ \square\ Ps
\end{array}$$

On top of these, we have formalised the standard concepts of pure, intuitionistic, and precise formulas together with their main properties. We note to Isabelle that separating conjunction forms a commutative, additive monoid with the empty heap assertion. This means all library properties proved about this structure become automatically available, including laws about fold over lists of assertions.

From this development, we can set up standard simplification rule sets, such as maximising quantifier scopes (which is the more useful direction in separation logic), that are directly applicable in instances.

The assertions we cannot formalise on this abstract level are maps-to predicates such as the classical $p \mapsto v$. These depend on the underlying structure and can only be done on at least a partial instantiation.

Future work for the abstract development could include a second layer introducing assumptions on the semantics of the programming language instance. It then becomes possible to define locality, the frame rule, and (best) local actions generically for those languages where they make sense, e.g. for deep embeddings.

3 Automation

This section gives a brief overview of the automated tactics we have introduced on top of the abstract separation algebra formalisation.

There are three main situations that make interactive mechanical reasoning about separation logic in HOL frameworks cumbersome. Their root cause is that the built-in mechanism for managing assumption contexts does not work for the substructural separation logic and therefore needs to be done manually.

The first situation is the application of simple implications and the removal of unnecessary context. Consider the goal $(P \wedge* p \mapsto v \wedge* Q)\ h \Longrightarrow (Q \wedge* P \wedge* p \mapsto -)\ h$. This should be trivial and automatic, but without further support requires manual rule applications for commutativity and associativity of $\wedge*$ before the basic implication between $p \mapsto v$ and $p \mapsto -$ can be applied. Rewriting with AC rules alleviates the problem somewhat, but leads to unpleasant side effects when there are uninstantiated schematic variables in the goal. In a normal, boolean setting, we would merely have applied the implication as a forward rule and solved the rest by assumption, having the theorem prover take care of reordering, unification, and assumption matching.

While in a substructural logic, we cannot expect to always be able to remove context, at least the order of conjuncts should be irrelevant. We expect to apply a rule of the form $(P \wedge* Q)\ h \Longrightarrow (P' \wedge* Q)\ h$ either as a forward, destruction, or introduction rule where the real implication is between P and P' and Q tells us it can be applied in any context. Our tactics *sep_frule*, *sep_drule*, and *sep_rule* try rotating assumptions and conclusion of the goal respectively until the rule matches. If P occurs as a top-level separation conjunct in the assumptions, this will be successful, and the rule is applied. This takes away the tedium of positional adjustments and gives us basic rule application similar to plain HOL. The common case of reasoning about heap updates falls into this category. Heap update can be characterised by rules such as $(p \mapsto - \wedge* Q)\ h \Longrightarrow (p \mapsto v \wedge* Q)\ (h(p \mapsto v))$ if h is a simple heap map. If we encounter a goal with an updated heap $h(p \mapsto v)$ over a potentially large separating conjunction that mentions the term $p \mapsto v$, we can now make progress with a simple *sep_rule* application.

Note that while the application scenario is instance dependent, the tactic is not. It simply takes a rule as parameter.

The second situation is reasoning about heap values. Again, consider a simple heap instantiation of the framework. The rule to apply would be $(p \mapsto v \wedge* Q)\ h \Longrightarrow \text{the}\ (h\ p) = v$. The idea is similar to above, but this time we extend Isabelle's substitution tactic to automatically solve the side condition of the substitution by rotating conjuncts appropriately after applying the equality. It is important for this to happen atomically to the user, because the equality will instantiate the rule only partially in h and p, while the side condition determines the value v. Again, the tactic is generic, the rule comes from the instantiation.

The third situation is clearing context to focus on the interesting implication parts of a separation goal after heap update and value reasoning are done. The idea is to automatically remove all conjuncts that are equal in assumption and conclusion as well as solve any trivial implications. The tactic *sep_cancel* that achieves this is higher-level than the tactics above, building on the same principles.

Finally, we supply a low-level tactic *sep_select n* that rotates conjunct *n* to the front while leaving the rest, including schematics, untouched.

With these basic tactics in place, higher-level special-purpose tactics can be developed much more easily in the future. The rule application and substitution tactics fully support backtracking and chaining with other Isabelle tactics.

One concrete area of future work in automation is porting Kolanski's machinery for automatically generating mapsto-arrow variants [5], e.g. automatically lifting an arrow to its weak variant, existential variant, list variant, etc, including generating standard syntax and proof rules between them which could then automatically feed into tools like *sep_cancel*. Again, the setup would be generic, but used to generate rules for instances.

4 Instantiations

We have instantiated the library so far to four different languages and variants of separation logic.

For the first instance, we took an existing example for separation logic that is part of the Isabelle distribution and ported it to sit on top of the library. The effort for this was minimal, less than an afternoon for one person, and unsurprisingly the result was drastically smaller than the original, because all boilerplate separation logic definitions and syntax declarations could be removed. The example itself is the classic separation logic benchmark of list reversal in a simple heap of type *nat* $\Rightarrow$ *nat option* on a language with a VCG, deeply embedded statements and shallowly embedded expressions.

The original proof went through almost completely unchanged after replacing names of definitions. We then additionally shrunk the original proof from 24 to 14 lines, applying the tactics described above and transforming the script from technical details to reasoning about semantic content.

While a nice first indication, this example was clearly a toy problem. A more serious and complex instance of separation logic is a variant for reasoning about virtual memory by Kolanski [5]. We have instantiated the library to this variant as a test case, and generalised the virtual memory logic in the process.

The final two instantiations are taken from the ongoing verification of user- and kernel-level system initialisation in the seL4 microkernel [3]. Both instantiations are for a shallow embedding using the nondeterministic state monad, but for two different abstraction levels and state spaces. In the more abstract, user-level setting, the overall program state contains a heap of type *obj_id* $\Rightarrow$ *obj option*, where *obj* is a datatype with objects that may themselves contain a map *slot* $\Rightarrow$ *cap option* as well as further atomic data fields. In this setting we would like to not just separate parts of the heap, but also separate parts of the maps within objects, and potentially their fields. The theoretical background of this is not new, but the technical implementation in the theorem prover is nontrivial. The clear interface of separation algebra helped one of the authors with no prior experience in separation logic to instantiate the framework within a week, and helped an undergraduate student to prove separation logic specifications of seL4 kernel functions within the space of two weeks. This is a significant improvement over previous training times in seL4 proofs.

The second monadic instance is similar in idea to the one above, but technically more involved, because it works on a lower abstraction level of the kernel, where in-object maps are not partial, some of the maps are encoded as atomic fields, and the data types are more involved. To use these with separation logic, we had to extend the state space by ghost state that encodes the partiality demanded by the logic. The instantiation to the framework is complete, and we can now proceed with the same level of abstraction in assertions as above.

Although shallow embeddings do not directly support the frame rule, we have found the approach of baking the frame rule into assertions by appending `∧* P` to pre- and post-conditions productive. This decision is independent of the library.

5 Conclusion

We have presented early work on a lightweight Isabelle/HOL library with an abstract type class for separation algebra and generic support for interactive separation logic tactics. While we have further concrete ideas for automation and for more type class layers with deeper support of additional separation logic concepts, the four nontrivial instantiations with productive proofs on top that we could produce in a short amount of time show that the concept is promising.

The idea is to provide the basis for rapid prototyping of new separation logic variants on different languages, be they deep or shallow embeddings, and of new automated interactive tactics that can be used across a number of instantiations.

Acknowledgements. We thank Matthias Daum for his comments on this paper.

References

1. Bengtson, J., Jensen, J., Sieczkowski, F., Birkedal, L.: Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 22–38. Springer, Heidelberg (2011)
2. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proc. 22nd LICS, pp. 366–378. IEEE (2007)
3. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSP, pp. 207–220. ACM (October 2009)
4. Klein, G., Kolanski, R., Boyton, A.: Separation algebra. Archive of Formal Proofs (May 2012), http://afp.sf.net/entries/Separation_Algebra.shtml
5. Kolanski, R.: Verification of Programs in Virtual Memory Using Separation Logic. PhD thesis, School Comp. Sci. & Engin. (July 2011)
6. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) 34th POPL, Nice, France, pp. 97–108. ACM (January 2007)
7. Tuerk, T.: A Formalisation of Smallfoot in HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 469–484. Springer, Heidelberg (2009)

Directions in ISA Specification

Anthony Fox

Computer Laboratory, University of Cambridge, UK

Abstract. This rough diamond presents a new domain-specific language (DSL) for producing detailed models of Instruction Set Architectures, such as ARM and x86. The language's design and methodology is discussed and we propose future plans for this work. Feedback is sought from the wider theorem proving community in helping establish future directions for this project. A parser and interpreter for the DSL has been developed in Standard ML, with an ARMv7 model used as a case study.

This paper describes recent work on developing a domain-specific language (DSL) for Instruction Set Architecture (ISA) specification. Various theorem proving projects require ISA models; for example, for formalizing microprocessors, operating systems, compilers and machine code. As such, (often partial) ISA models exist for a number of architectures (e.g. x86, ARM and PowerPC) in a number of theorem provers (e.g. ACL2, PVS, HOL-Light, Isabelle/HOL, Coq and HOL4). These models differ in their presentation style, precise abstraction level (fidelity) and degrees of completeness. In part this reflects the nature of the projects for which the models have been originally developed, e.g. compiler verification [4] and machine code verification [7]. There are also differences based on the expressiveness and features of the theorem provers that are used. The ACL2 theorem prover has been used very successfully in this field for many years, where it has the advantage of providing very fast model evaluation. Recently, Warren Hunt has developed an ACL2-based specification of the Y86 processor, which implements a subset of the x86 architecture; see [3].

The main objective of the DSL is to make the task of modelling ISAs simpler, more reliable and less tedious. In particular, it should be possible for people who are not experts in HOL4 to readily read, develop and create ISA specifications for use in HOL4. Furthermore, it is also hoped that this work will help facilitate the dissemination of ISA models — enabling various concrete ISA models to be derived for different settings, tools and use cases.

Although various ISA DSLs currently exist, often these have been developed for writing compiler backends and binary code analysis tools, e.g. λ-RTL [9] and TSL [5]. The most closely related work is Lyrebird [1], which was developed as part of the seL4 project at NICTA. This tool supports fast simulation but it has not been successfully used in a theorem proving setting. Also of interest is the RockSalt project [6], which modelled x86 in Coq through the use of embedded DSLs. The aim of this work is to produce high-fidelity specifications that are inherently formal and yet prover/tool agnostic. The DSL and generated native prover specifications should be acceptable to both the engineering (computer architecture) and formal methods communities.

1 Language Design and Methodology

The design of the DSL has been influenced by our experiences in specifying the ARM architecture in HOL4, which is described in [2]. In particular, the DSL has been developed and tested through the production of a completely new version of the ARMv7 specification.[1] However, it is believed that the DSL is flexible enough to produce good models of other ISAs, such as the x86 architecture.

Methodology. The requirements for the DSL are based on our current specification approach, where we define:

– A *state space*. This represents all of the programmer visible registers, flags and memory. It may also include components that are not directly visible, such as static system configuration information (e.g. describing the architecture version and extension support) as well as any helpful shadow state components (e.g. the bit width of the current instruction).
– An *instruction datatype*. This provides an interface between instruction decoders and the instruction set semantics.
– A collection of definitions, specifying the semantics of each instruction class. This provides an operational (next state) semantics for each element of the instruction datatype.
– A *decoder*. This maps machine code values to the instruction datatype.
– An *encoder* (optional). This maps elements of the instruction datatype to concrete machine code values.
– A top-level next state function. This fetches an instruction from memory, decodes it and then applies the appropriate semantics definition for that instruction.

This approach has been implemented directly in HOL4 for the ARM, x86 and PowerPC architectures. However, there are areas where producing and maintaining ISA specifications in HOL4 is unduly tedious and potentially error prone.

The DSL improves upon native HOL4-based specifications in a few key areas:

– In HOL4 the state space is declared as a type, which means that all state components must be introduced early on and all in one go. It is also necessary to manually introduce collections of functions for accessing and updating state components and sub-components (e.g. named bit-fields).

  It is more natural to introduce state components in context, as and when they are needed. For example, within separate sections for the specification of machine registers and main memory. In the DSL state components are treated as global variables that may be declared anywhere at the top-level.

[1] The new model actually covers the very latest incarnation of ARMv7, which adds support for a new “hypervisor” mode.

- The instruction datatype is also declared as a HOL4 type, which is somewhat tedious to specify and to maintain.

  The instruction datatype can be built incrementally, using the type signatures of the functions that define the instruction semantics.
- Writing a good decoder is particularly challenging in HOL4. This is primarily because HOL4 does not provide direct support for matching over bit patterns. There is also the challenge of making the decoder evaluate efficiently, which currently requires some degree of HOL4 expertise.

  Matching over bit patterns is built into the DSL. We hope to support efficient evaluation for the generated HOL4 model.

These and other language features make it much easier to write ISA specifications in a natural style; making it possible to automatic generate HOL4 specifications that are otherwise hard or tedious to write manually.

Language Overview. The DSL is a first-order language with a fairly basic type system.[2] The intention is to keep the design of the DSL reasonably simple; shunning features that are not directly focussed on the ISA domain. This reduces the effort required to implement the language and it should help simplify the task of targeting models to different settings. Although the DSL is not particularly sophisticated, there were no problems in concisely specifying ARMv7.

Types. The primitive types of the language are: `unit`, `bool`, `string`, `nat`, `int`, `bitstring` and `bits(n)`, where `n` is either fixed or is constrained to a (possibly infinite) set of positive integers at the point of a function definition.[3] Type checking is implemented using Hindley-Milner inference, with some additional light-weight support for bit-vectors. Users can declare type synonyms, records, enumerations and non-recursive sum types. Constructors for product, map and set types are provided. For example, the following are valid declarations:

```
type reg  = bits(4)              -- type synonym (this is a comment)
type mem  = bits(32) → bits(8)   -- map

construct SRType -- enumerated type
{ SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX }

construct offset -- sum type
{  register_form  :: reg * SRType * nat  -- product
   immediate_form :: bits(32)  }
```

Syntax and Constructs. The DSL syntactically distinguishes between statements and expressions. Mutable values can be declared and updated in statements but not in expressions. There are *if-then-else*, *when-do*, *match-case* and *for-do* constructs. Function calls are strictly call-by-value but side-effects are possible,

[2] Recursive types, type polymorphism and dependent types are not supported.

[3] Floating-point support will be added in the future.

i.e. the global state can be updated. Exceptions can be declared and called, but not handled. A wide selection of primitive data operations are provided.

Users can define their own operations but cannot give these symbolic or infix/mixfix syntax. The following declaration defines n-byte word alignment:

```
bits(N) Align (w::bits(N), n::nat) = return [n * ([w] div n)]
```

The operation '`[·]`' is used as a general casting map for primitive types. All types are inferred above using the function's arguments, which must be annotated.

Registers. The DSL supports declarations of register types with named bit-fields. The following declares a type for ARM's Programme Status Registers:

```
register PSR :: word
{  31: N  30: Z
   29: C  28: V      -- Negative, Zero, Carry, oVerflow flags
   27:    Q          -- Cumulative saturation flag
   15-10, 26-25: IT  -- If-Then
   24:    J          -- Jazelle bit
   23-20: RAZ!       -- reserved
   19-16: GE         -- Greater-equal flags (SIMD)
   9:     E          -- Endian bit (T: Big, F: Little)
   8:     A          -- Asynchronous abort disable
   7:     I          -- Interrupt disable
   6:     F          -- Fast interrupt disable
   5:     T          -- Thumb mode
   4-0:   M          -- Mode field  }
```

This introduces a new type `PSR` that corresponds with a 32-bit word. The named bit-field `M` is a 5-bit word, `N` is a Boolean flag and the special value `RAZ!` (read-as-zero) signifies an anonymous field. The expression `CPSR.IT` is equivalent to `&CPSR<15:10> : &CPSR<26:25>`; where the overloaded operator '`&`' maps registers to their bit-vector values, '`·<·:·>`' is bit-field extraction and '`:`' is bit-vector concatenation. In the HOL4 model [2], PSRs are defined using a record type and encoding/decoding functions are manually defined. It is now relatively easy to automatically generate these types and functions for each of ARM's system registers, saving users time and effort.

State. Global state components are declared as follows:

```
declare CPSR :: PSR
declare MEM  :: bits(32) → bits(8)
```

These components can be updated with various assignment forms, for example:

```
CPSR.N ← true;       &CPSR<31> ← true;        CPST.M ← '11010';
&CPSR ← 0x11;        &CPSR<31:28> ← '1101';   CPSR ← PSR(0x11);
MEM(4) ← &CPSR<15:8>
```

The dot syntax also applies to conventional record types. Users can define their own update operations; for example, consider the following declaration:

```
component NZCV :: bits(4)
{ value = &CPSR<31:28>
  assign v = &CPSR<31:28> ← v  }
```

This declaration makes it easy to access and modify the NZCV status flags. The component construct is particularly useful for declaring operations that access registers and memory. For example, one can specify:

```
NZCV ← NZCV && '0101';
R(12)<15:0> ← MemU(address, 2);
MemU(address + 2, 2) ← R(12)<31:16>
```

where the operations `R` and `MemU` provide an interface to the general-purpose registers and memory. Note that the physical register corresponding with the argument 12 actually depends on the current processor mode, given by `CPSR.M`; and memory accesses are affected by the endian bit `CPSR.E`.

Instruction Specification. The following DSL code specifies the semantics of the ARM instruction BLX (register):

```
define Branch > BranchLinkExchangeRegister ( m :: reg ) =
{  target = R(m);
   if CurrentInstrSet() == InstrSet_ARM then
   {  next_instr_addr = PC - 4;
      LR ← next_instr_addr
   } else
   {  next_instr_addr = PC - 2;
      LR ← next_instr_addr<31:1> : '1'
   };
   BXWritePC (target)
}
```

This declaration extends an *abstract syntax tree* (AST) datatype `instruction`. A primitive operation `Run :: instruction → unit` runs the code associated with the given AST. The > notation allows instructions to be grouped into a hierarchy of instruction categories.

Instruction Decoding. A decoder is any function that takes the output of an instruction fetch and returns values of type `instruction`. Users are free to define such functions in any way that they see fit. A natural choice for decoding ARM instructions is through pattern matching over bit patterns. The ARMv7 decoder is approximately four thousand lines of code (including comments), with 233 top-level cases. Missing and redundant patterns are reported, which is essential in this context. Below is a code snippet relating to the BLX instruction:

```
instruction DecodeARM (w::bits(32)) =
  match w
  { ...
    case 'cond : 00010010 : (111111111111) : 0011 : Rm' =>
    if Take (cond, ArchVersion() >= 5) then
    {  when Rm == 15 do DECODE_UNPREDICTABLE (mc, "BLX (register)");
       Branch (BranchLinkExchangeRegister (Rm))
    } else Skip ()
    ...
  }
```

Bit patterns are surrounded by apostrophes. Bracketed bit fields are "should-be" tokens — they match *any* field of the appropriate length. The bit-widths of variables can be annotated or given default values: `cond` and `Rm` were declared as 4-bit values. When an op-code is not valid the user function `DECODE_UNPREDICTABLE` is called, which raises a suitable exception. The user defined functions `Take` and `Skip` take care of conditional (no-op) and undefined instructions.

2 Directions

At the time of writing the new DSL has a parser and an evaluator/interpreter. Good progress has been made on exporting to HOL4 — the initial objective is to generate a first-order functional specification for ARMv7. One of the advantages of having a custom DSL is that different models can be generated from the same source. For example, it should be possible to generate deeply embedded models or monadic specifications similar to those already in HOL4. One key objective is to generate "evaluation friendly" code, i.e. to support fast evaluation for machine code verification (see [2]).

The back-end model export should be readily configurable, facilitating translations into various other settings, such as different theorem provers (ACL2, Coq, HOL-Light and Isabelle/HOL), high-level languages (e.g. SML and Bluespec) or OpenTheory.[4] This would enable the same model to be used in wide range of projects, potentially avoiding duplicated effort. In this regard our objective is similar to that of Lem [8]. A key challenge will be to generate satisfactory models (meeting the requirements of end-users) under each setting. Our main priority is to improve our own, well-understood HOL4 workflow. In targeting other theorem provers, dialogue and collaboration with other end-users will be essential.

Acknowledgments. Many thanks to Magnus Myreen, Mike Gordon and Peter Sewell for providing motivation and stimulating discussions.

[4] `http://www.gilith.com/research/opentheory`

References

1. Cock, D.: Lyrebird: assigning meanings to machines. In: SSV 2010 (2010)
2. Fox, A., Myreen, M.O.: A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010)
3. Hunt Jr., W.A.: X86 specification in ACL2, http://www.cs.utexas.edu/~hunt/research/y86/
4. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4) (2009)
5. Lim, J., Reps, T.: A System for Generating Static Analyzers for Machine Instructions. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 36–52. Springer, Heidelberg (2008)
6. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: Better, faster, stronger SFI for the x86. In: PLDI 2012 (2012)
7. Myreen, M.O., Gordon, M.J.C.: Verified LISP Implementations on ARM, x86 and PowerPC. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 359–374. Springer, Heidelberg (2009)
8. Owens, S., Böhm, P., Zappa Nardelli, F., Sewell, P.: Lem: A Lightweight Tool for Heavyweight Semantics. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 363–369. Springer, Heidelberg (2011)
9. Ramsey, N., Davidson, J.W.: Machine Descriptions to Build Tools for Embedded Systems. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 176–192. Springer, Heidelberg (1998)

More SPASS with Isabelle

Superposition with Hard Sorts and Configurable Simplification

Jasmin Christian Blanchette[1], Andrei Popescu[1],
Daniel Wand[2], and Christoph Weidenbach[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. Sledgehammer for Isabelle/HOL integrates automatic theorem provers to discharge interactive proof obligations. This paper considers a tighter integration of the superposition prover SPASS to increase Sledgehammer's success rate. The main enhancements are native support for hard sorts (simple types) in SPASS, simplification that honors the orientation of Isabelle *simp* rules, and a pair of clause-selection strategies targeted at large lemma libraries. The usefulness of this integration is confirmed by an evaluation on a vast benchmark suite and by a case study featuring a formalization of language-based security.

1 Introduction

The interactive theorem proving community has traditionally put more emphasis on trustworthiness, expressiveness, and flexibility than on raw deductive power. Automation in proof assistants typically takes the form of general-purpose proof methods or tactics, complemented by decision procedures for specific domains. Recent large-scale efforts such as the proofs of the four color theorem [16], of a C compiler [23], and of a microkernel [20] have highlighted the need for more automation [24].

There have been many attempts at harnessing decades of research in automated reasoning by integrating automatic theorem provers in proof assistants. The most successful integration is undoubtedly Sledgehammer for Isabelle/HOL (Sects. 2.1 and 2.2). The tool invokes several first-order automatic provers in parallel, both superposition-based provers and SMT solvers [5], and reconstructs their proofs in Isabelle. In an evaluation on a representative corpus of older formalizations, Sledgehammer discharged 43% of the goals that could not be solved trivially using an existing Isabelle proof method [5].

Sledgehammer's usefulness is regularly confirmed by users; Guttmann et al. [17] relied almost exclusively on it to derive over 1000 propositions relating to relational and algebraic methods for modeling computing systems. Yet, there are many indications that more can be done. Integrated verification tool chains such as VCC/Boogie/Z3 [12] claim much higher success rates, typically well above 90%. Isabelle goals are certainly more diverse than verification conditions for a fixed programming language, which makes fine-tuning less practicable; however, typical Isabelle goals are not necessarily more difficult than typical verification conditions. Another indicator that Sledgehammer can be significantly improved is that it solves only about 80% of the goals that standard proof methods (such as *simp*, *auto*, and *blast*) solve trivially [5]. This lackluster performance points to weaknesses both in Sledgehammer's translation of higher-order constructs and in the automatic provers themselves.

What is easy in a proof assistant can be surprisingly difficult for first-order automatic theorem provers. For example, Isabelle's simplifier can easily prove goals of the form rev $[x_1,\ldots,x_n] = [x_n,\ldots,x_1]$ (where rev is list reversal) by applying equations as oriented rewrite rules, or *simp* rules. The presence of hundreds of registered *simp* rules hardly slows it down. In contrast, superposition provers such as E [38], SPASS [44], and Vampire [34] perform an unconstrained search involving the supplied background theory. They can use equations as rewrite rules but must often reorient them to obey a specific term ordering. In exchange, these provers are complete for classical first-order logic: Given enough resources, they eventually find a (first-order) proof if one exists.

Much work went into making Sledgehammer's translation from higher-order logic as efficient as possible [6,27]. However fruitful this research may have been, it appears to have reached a plateau. To achieve higher success rates, a new approach is called for: Implement the features that make proof search in Isabelle successful directly in an automatic prover. By combining the best of both worlds, we target problems that cannot be solved by either tool on its own.

Our vehicle is SPASS (Sect. 2.3), a widely used prover based on a superposition calculus (a generalization of resolution). It is among the best performing automatic provers, and one of the few whose development is primarily driven by applications, including mission-critical computations in industry. Its source code is freely available and well-structured enough to form a suitable basis for further development. In the automated reasoning community, SPASS is well-known for its soft sorts, which can comfortably accommodate many-sorted, order-sorted, and membership logics, its integrated splitting rule, employing competitive decision procedures for various decidable first-order fragments, and its sophisticated simplification machinery.

This paper describes the first part of our work program. It focuses on three aspects.

- *Hard sorts* (Sect. 3): Soft sorts are overly general for most applications. By supporting a more restrictive many-sorted logic in SPASS and combining it with monomorphization, we get a sound and highly efficient representation of HOL types, without the spurious unreconstructible proofs that have long plagued Sledgehammer users.
- *Configurable simplification* (Sect. 4): Superposition provers reorient the equations in the problem to make the right-hand sides smaller than the left-hand sides with respect to a term ordering. *Advisory* simplification adjusts the ordering to preserve the orientation of *simp* rules as much as possible; *mandatory* simplification forcefully achieves rewriting against the term ordering if necessary.
- *Clause selection for large theories* (Sect. 5): Sledgehammer problems include hundreds of lemmas provided as axioms, sorted by likely relevance. It makes sense for SPASS to focus on the most likely relevant lemmas, rather than hopelessly try to saturate the entire background theory.

A case study demonstrates the SPASS integration on a formalization pertaining to language-based security (Sect. 6), an area that calls for strong automation in combination with the convenience of a modern proof assistant. The new features are evaluated both in isolation and against other popular automatic theorem provers (Sect. 7).

2 Background

2.1 Isabelle/HOL

Isabelle/HOL [30] is a proof assistant based on classical higher-order logic (HOL) extended with rank-1 polymorphism and axiomatic type classes. The term language consists of simply-typed λ-terms augmented with constants (of scalar or function types) and polymorphic types. Function application expects no parentheses around the arguments (e.g., $f\ x\ y$); familiar operators are written in infix notation. Functions may be partially applied (curried), and variables may range over functions. Types can optionally be attached to a term using an annotation $t : \sigma$ to guide type inference.

The dominant general-purpose proof method is the simplifier, which applies equations as oriented rewrite rules to rewrite the goal. It performs conditional, contextual rewriting with hooks for customizations based on the vast library of registered *simp* rules. At the user level, the simplifier is upstaged by *auto*, a method that interleaves simplification with proof search. Other commonly used methods are the tableau prover (*blast*) and the arithmetic decision procedures (*linarith* and *presburger*).

2.2 Sledgehammer

Sledgehammer [31] harnesses the power of superposition provers and SMT solvers. Given a conjecture, it heuristically selects a few hundred facts (lemmas, definitions, or axioms) from Isabelle's libraries [28], translates them to first-order logic along with the conjecture, and delegates the proof search to external provers—by default, E [38], Vampire [34], Z3 [29], and of course SPASS [44]. Because automated deduction involves a large share of heuristic search, the combination of provers is much more effective than any single one of them. Proof reconstruction relies on the built-in resolution prover *metis* [19,32] and the Z3-based *smt* proof method [11].

Given that automatic provers are very sensitive to the encoding of problems, the translation from higher-order logic to unsorted first-order logic used for E, SPASS, and Vampire is a crucial aspect of Sledgehammer. It involves two steps [27]:

1. *Eliminate the higher-order features of the problem.* Curried functions are passed varying numbers of arguments using a deeply embedded application operator, and λ-abstractions are rewritten to SK combinators or supercombinators (λ-lifting).
2. *Encode polymorphic types and type classes.* Type information can be encoded in a number of ways. Traditionally, it has been supplied as explicit type arguments to the function and predicate symbols corresponding to HOL constants. In conjunction with Horn clauses representing the type class hierarchy, this suffices to enforce correct type class reasoning and overload resolution, but not to prevent ill-typed variable instantiations. Unsound proofs are discarded at reconstruction time.

Example 1. In the recursive specification of map on lists, the variable $f : \alpha \to \beta$ is higher-order both in virtue of its function type and because it occurs partially applied:

$$\mathsf{map}\ f\ \mathsf{Nil} = \mathsf{Nil}$$
$$\mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

Step 1 translates the second equation to

$$\mathsf{map}(F, \mathsf{cons}(X, Xs)) = \mathsf{cons}(\mathsf{app}(F, X), \mathsf{map}(F, Xs))$$

where $F : \mathit{fun}(\alpha, \beta)$ is a deeply embedded function (or "array") and app is the embedded application operator (or "array read").[1] Step 2 introduces type arguments encoded as terms, with term variables A, B for α, β:

$$\mathsf{map}(A, B, F, \mathsf{cons}(A, X, Xs)) = \mathsf{cons}(B, \mathsf{app}(A, B, F, X), \mathsf{map}(A, B, F, Xs))$$

2.3 SPASS

SPASS (= Spaß = "fun" in German) is an implementation of the superposition calculus with various refinements, including unique support for soft (monadic) sorts and splitting [15, 43, 44]. It is a semi-decision procedure for classical first-order logic and a decision procedure for various first-order logic fragments.

The input is a list of axioms and a conjecture expressed either in the TPTP FOF syntax [40] or in a custom syntax called DFG. SPASS outputs either a proof (a derivation of the empty, contradictory clause from the axioms and the negated conjecture) or a saturation (an exhaustive list of all normalized clauses that can be derived); it may also diverge for unprovable problems with no finite saturation.

Well-founded term orderings are crucial to the success of the superposition calculus. For example, from the pair of clauses $\mathsf{p}(\mathsf{a})$ and $\neg\,\mathsf{p}(X) \vee \mathsf{p}(\mathsf{f}(X))$, resolution will derive infinitely many facts of the form $\mathsf{p}(\mathsf{f}^i(\mathsf{a}))$, whereas for superposition $\mathsf{p}(\mathsf{f}(X))$ is maximal and no inferences can be performed. Nonetheless, superposition-based reasoning is very inefficient when combined with order-sorted signatures, because completeness requires superposition into variables, which dramatically increases the search space. Soft sorts were designed to remedy this problem: When they are enabled, SPASS views every monadic predicate as a sort and applies optimized inference and simplification rules [15]. Monadic predicates can be used to emulate a wide range of type systems.

3 Hard Sorts

After eliminating the higher-order features of a problem, Sledgehammer is left with first-order formulas in which Isabelle's polymorphism and axiomatic type classes still occupy a prominent place. The type argument scheme presented in Section 2.2 is unsound, and the traditional sound encodings of polymorphic types introduce too much clutter to be useful [6, 10, 27]. This state of affairs is unsatisfactory. Even with proof reconstruction, there are major drawbacks to unsound type encodings.

First, finite exhaustion rules of the form $x = \mathsf{c}_1 \vee \cdots \vee x = \mathsf{c}_n$ or $(x = \mathsf{c}_1 \Longrightarrow P) \Longrightarrow \cdots \Longrightarrow (x = \mathsf{c}_n \Longrightarrow P) \Longrightarrow P$ must be left out because they force an upper bound on the cardinality of the universe, rapidly leading to unsound cardinality reasoning; for example, automatic provers can easily derive a contradiction from $(u : \mathit{unit}) = ()$ and $(0 : \mathit{nat}) \neq \mathsf{Suc}\ n$ if type information is simply omitted. The inability to encode such rules prevents the discovery of proofs by case analysis on finite types.

[1] Following a common convention in the automated reasoning and logic programming communities, we start first-order variable names with an upper-case letter, keeping lower-case for function and predicate symbols. Nullary functions (constants) are written without parentheses.

Second, spurious proofs are distracting and sometimes conceal more difficult sound proofs. Users eventually learn to recognize facts that lead to unsound reasoning and mark them with a special attribute to remove them from the scope of Sledgehammer's relevance filter, but this remains a stumbling block for novices.

Third, it would be desirable to let SPASS itself perform relevance filtering, or even use a sophisticated system based on machine learning, where successful proofs guide subsequent ones. However, such approaches tend to quickly detect and exploit contradictions in the large translated axiom set if type information is omitted.

How can we provide SPASS with the necessary type information in a sound, complete, and efficient manner? The original plan was to exploit SPASS's soft sorts, by monomorphizing the problem (i.e., heuristically instantiating the type variables with ground types) and inserting monadic predicates, or guards, $\mathsf{p}_\sigma(X)$ to ensure that a given variable X has type σ. Following this scheme, the *nat* $\rightarrow$ *int* instance of the second equation in Example [1] would be translated to the unsorted formula

$$\begin{aligned}&\mathsf{p}_{\mathsf{fun(nat,\,int)}}(F) \wedge \mathsf{p}_{\mathsf{nat}}(X) \wedge \mathsf{p}_{\mathsf{list(nat)}}(Xs) \longrightarrow\\&\qquad \mathsf{map}_{\mathsf{nat,\,int}}(F, \mathsf{cons}_{\mathsf{nat}}(X, Xs)) = \mathsf{cons}_{\mathsf{int}}(\mathsf{app}_{\mathsf{nat,\,int}}(F, X), \mathsf{map}_{\mathsf{nat,\,int}}(F, Xs))\end{aligned}$$

where subscripts distinguish instances of polymorphic symbols. The guards are discharged by deeply embedded typing rules for the function symbols occurring in the problem. SPASS views each p_σ predicate as a sort.

Monomorphization is necessarily incomplete [9, §2] and often dismissed because it quickly leads to an explosion in the number of formulas. Nonetheless, with suitable bounds on the number of monomorphic instances generated, our experience is that it vastly outperforms complete encodings of polymorphism [6]. It also relieves SPASS of having to reason about type classes—only the monomorphizer needs to consider them.

The outcome of experiments with SPASS quickly dashed our hopes: Sure enough, soft sorts were helping SPASS, but the resulting encoding was still no match for the unsound scheme based on type arguments. Explicit typing requires a particular form of contextual rewriting to simulate typed rewriting efficiently. The needed mechanisms are not available in any of today's first-order theorem provers, not even in SPASS's soft typing machinery. For example, consider a constant a of sort σ and an unconditional equation $\mathsf{f}(X) = X$ where $X : \sigma$. Sorted rewriting transforms $\mathsf{f}(\mathsf{a})$ into a in one step. In contrast, the soft typing version of the example is a conditional equation $\mathsf{p}_\sigma(X) \longrightarrow \mathsf{f}(X) = X$ and the typing axiom $\mathsf{p}_\sigma(\mathsf{a})$. Rewriting $\mathsf{f}(\mathsf{a})$ requires showing $\mathsf{p}_\sigma(\mathsf{a})$ to discharge the condition in the instance $\mathsf{p}_\sigma(\mathsf{a}) \longrightarrow \mathsf{f}(\mathsf{a}) = \mathsf{a}$.

We came up with a new plan: Provide *hard* sorts directly in SPASS, orthogonally to soft sorts. Hard sorts can be checked directly to detect type mismatches early and avoid ill-sorted inferences. We focused on monomorphic sorts, which require no matching or unification. The resulting many-sorted first-order logic corresponds to that offered by the TPTP TFF0 format [42]. Polymorphism is eliminated by monomorphization.[2]

[2] For future work, we want to extend the hard sorts to ML-style polymorphism as provided by Alt-Ergo [8] and TPTP TFF1 [7]. Besides the expected performance benefits [14], this is a necessary step toward mirroring Isabelle's hierarchical theory structure on the SPASS side: Once theories are known to SPASS, they can be preprocessed (e.g., finitely saturated) and reused, which would greatly speed up subsequent proof searches.

Although superposition with hard sorts is well understood, adding sorts to a highly optimized theorem prover is a tedious task. Predicate and function symbols must be declared with sort signatures. Variables must carry sorts, and unification must respect them. The term index that underlies most inference rules must take sort constraints into consideration; the remaining rules must be adapted individually. There are many other technical aspects related to variable renaming, skolemization, and of course parsing and printing of formulas. We made all these changes and found that hard sorts are much more efficient than their soft cousins, and even than the traditional unsound scheme, which we so desperately wanted to abolish.

In a fortuitous turn of events, a group of researchers including the first author recently discovered a lightweight yet sound guard-based encoding as well as many variants [6]. These are now implemented in Sledgehammer. They work well in practice but fall short of outperforming hard sorts. Moreover, hard sorts are more suitable for applications that require not only soundness of the overall result but also type-correctness of the individual inferences, such as step-by-step proof replay [32].

4 Configurable Simplification

The superposition calculus is parameterized by a well-founded total ordering on ground terms. Like most other provers, SPASS employs the Knuth–Bendix ordering [22], which itself is determined by a weight function w from symbols to $\mathbb{N}$ and a total precedence order $\prec$ on function symbols. Weights are lifted to terms by taking the sum of the weights of the symbols that occur in it, counting duplicates.

Let $s = \mathsf{f}(s_1,\ldots,s_m)$ and $t = \mathsf{g}(t_1,\ldots,t_n)$ be two terms. The (*basic*) *Knuth–Bendix ordering* (*KBO*) induced by $(w,\prec)$ is the relation $\prec$ such that $s \prec t$ if and only if for any variable occurring in s it has at least as many occurrences in t and a, b, or c is satisfied:

a. $w(s) < w(t)$;
b. $w(s) = w(t)$ and $\mathsf{f} \prec \mathsf{g}$;
c. $w(s) = w(t)$, $\mathsf{f} = \mathsf{g}$, and there exists i such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$, and $s_i \prec t_i$.

Assuming w and $\prec$ meet basic requirements, the corresponding KBO is a well-founded total order on ground terms that embeds the subterm relation and is stable under substitution, as required by superposition. The main proviso for the application of an equation $l = r$ to simplify a clause is that l must be larger than r with respect to the given KBO.

By default, SPASS simply assigns a weight of 1 to every symbol and heuristically selects a precedence order. Then it reviews each equation $l = r$ in the light of the induced KBO to determine whether $l = r$, $r = l$, or neither of them can be applied as a left-to-right rewrite rule to simplify terms. Since the left-hand side of an Isabelle definition tends to be smaller than the right-hand side, SPASS will often reorient definitions, making it much more difficult to derive long chains of equational reasoning.

Intuitively, a better strategy would be to select a weight function and a precedence order that maximize the number of definitions and *simp* rules that SPASS can use for simplification with their original orientation. For example, to keep the equation

$$\mathsf{shift}(\mathsf{cons}(X, Xs)) = \mathsf{append}(Xs, \mathsf{cons}(X, \mathsf{nil}))$$

oriented, SPASS could take $w(\mathsf{shift}) \geq 3$ (or even $w(\mathsf{shift}) = 2$ with $\mathsf{append} \prec \mathsf{shift}$) while setting $w(\mathsf{nil}) = w(\mathsf{cons}) = w(\mathsf{append}) = 1$; the occurrences of X and Xs on either side cancel each other out. The weight function normally plays a greater role than the precedence order, but for some equations precedence is needed to break a tie—for example:

$$\mathsf{append}(\mathsf{cons}(X, Xs), Ys) = \mathsf{cons}(X, \mathsf{append}(Xs, Ys))$$

Our approach for computing a suitable weight function w is to build a dependency graph, in which edges $\mathsf{f} \leftarrow \mathsf{g}$ indicate that f is simpler than g. The procedure first considers definitional equations of the form $\mathsf{f}(s_1, \ldots, s_m) = t$, including simple definitions and equational specifications of recursive functions, and adds edges $\mathsf{f} \leftarrow \mathsf{g}$ for each symbol g that occurs in s_1, ..., s_m, or t, omitting any edge that would complete a cycle (which may happen if f is recursive through g). In a second step, *simp* rules are considered in the same way to further enrich the graph. The cycle-detection mechanism is robust enough to cope with nondefinitional lemmas such as $\mathsf{rev}(\mathsf{rev}(Xs)) = Xs$.

Once the graph is built, the procedure assigns weight 2^{d+1} to symbols with depth d and uses a topological order for symbol precedence. For example, given the usual recursive definitions of append (in terms of nil and cons) and rev (in terms of append, nil, and cons), it computes $w(\mathsf{nil}) = w(\mathsf{cons}) = 1$, $w(\mathsf{append}) = 2$, $w(\mathsf{rev}) = 4$, and either $\mathsf{nil} \prec \mathsf{cons} \prec \mathsf{append} \prec \mathsf{rev}$ or $\mathsf{cons} \prec \mathsf{nil} \prec \mathsf{append} \prec \mathsf{rev}$ for the precedence. With these choices of w and $\prec$, SPASS proves $\mathsf{rev}\ [x_1, \ldots, x_n] = [x_n, \ldots, x_1]$ in no time even for large values of n (e.g., 50) and in the presence of hundreds of axioms, whereas the other automatic provers time out.

Regrettably, there are many equations that cannot be oriented in the desired way with this approach. KBO cannot orient an equation such as

$$\mathsf{map}(F, \mathsf{cons}(X, Xs)) = \mathsf{cons}(\mathsf{app}(F, X), \mathsf{map}(F, Xs))$$

in a left-to-right fashion because of the two occurrences of F on the right-hand side. It will also fail with

$$\mathsf{rev}(\mathsf{cons}(X, Xs)) = \mathsf{append}(\mathsf{rev}(Xs), \mathsf{cons}(X, \mathsf{nil}))$$

because the occurrences of rev and cons on the left-hand side are canceled out by those on the right-hand side; no matter how heavy we make these, the right-hand side will weigh even more due to append's and nil's contributions.

An especially thorny yet crucial example is the S combinator, defined in HOL as $\lambda x\, y\, z.\ x\ z\ (y\ z)$. It manifests itself in most problems generated by Sledgehammer to encode λ-abstractions. In first-order logic, it is specified by the axiom

$$\mathsf{app}(\mathsf{app}(\mathsf{app}(\mathsf{s}, X), Y), Z) = \mathsf{app}(\mathsf{app}(X, Z), \mathsf{app}(Y, Z))$$

For simplification, the left-to-right orientation is clearly superior, because it eliminates the combinator whenever the third argument is supplied, emulating β-reduction. Unfortunately, the duplication of Z on the right-hand side makes this orientation incompatible with KBO; in fact, either orientation is incompatible with the subterm condition and substitution stability requirements on admissible term orderings.

All is not lost for equations that cannot be ordered in the natural way. It is possible to extend superposition with controlled simplification against the term ordering. To achieve this, we extended SPASS's input syntax with annotations for both *advisory* simplifications, which only affect the term ordering, and *mandatory* simplifications, which force rewriting against the ordering if necessary.

To avoid infinite looping, the mandatory simplification rules must terminate. Isabelle ensures that simple definitions have acyclic dependencies and recursive function specifications are well-founded, so these can safely be made mandatory. Artifacts of the translation to first-order logic, such as the SK combinators, can also be treated in this way. We could even trust the Isabelle user and make all *simp* rules mandatory, but it is safer to keep the advisory status for these. However, even assuming termination of mandatory simplifications, our implementation is generally incomplete; to ensure completeness, we would need to treat such simplifications as a separate inference rule of the superposition calculus, rather than as a postprocessing step.

Of course, excessive rewriting, especially of the mandatory kind, can give rise to large terms that hamper abstract reasoning. We encountered a striking example of this in the innocuous-looking HOL definitions $\mathsf{Bit0}\ k = k + k$ and $\mathsf{Bit1}\ k = 1 + k + k$, which together with Pls and Min (i.e., 0 and -1) encode signed numerals—for example, '4' is surface syntax for $\mathsf{Bit0}(\mathsf{Bit0}(\mathsf{Bit1}(\mathsf{Pls})))$. Rewriting huge numerals to sums of 1s is obviously detrimental to SPASS's performance, so we disabled mandatory simplification for these two definitions.

5 Clause Selection for Large Theories

Superposition provers work by exhaustively deriving all possible (normalized) clauses from the supplied axioms and the negated conjecture, aiming at deriving the empty clause. New clauses are produced by applying inference rules on already derived pairs of clauses. The order in which clauses are selected to generate further inferences is crucial to a prover's performance and completeness.

At the heart of SPASS is a set of *usable* (passive) clauses $\mathcal{U}$ and a set of *worked-off* (active) clauses $\mathcal{W}$ [43]. The set $\mathcal{U}$ is initialized with the axioms and negated conjecture, whereas $\mathcal{W}$ starts empty. The prover iteratively performs the following steps:

1. Heuristically select a clause C from $\mathcal{U}$.
2. Perform all possible inferences between C and each member of $\mathcal{W} \cup \{C\}$ and insert the resulting clauses into $\mathcal{U}$.
3. Simplify $\mathcal{U}$ and $\mathcal{W}$ using $\mathcal{W} \cup \{C\}$ and move C to $\mathcal{W}$.

A popular variant, the *set of support* (SOS) strategy [45], keeps the search more goal-oriented by initially moving the axioms into the worked-off set rather than into the usable set; only the negated conjecture is considered usable. This disables inferences between axioms, allowing only inferences that directly or indirectly involve the negated conjecture. SOS is complete for resolution but incomplete for superposition. It often terminates very fast, with either a proof or an incomplete saturation. A study found it advantageous for Sledgehammer problems [10, §3].

The heuristic that chooses a usable clause in step 1 is called the *clause-selection strategy*. SPASS's default strategy alternately chooses light clauses (to move toward the empty clause) and shallow clauses (to broaden the search) in a fair way. The *weight* of a clause is the sum of the weights of its terms (cf. Sect. 4); the *depth* is the height of its derivation tree. However, this strategy scales poorly with the number of facts provided by Sledgehammer; beyond about 150 facts (before monomorphization), additional facts harm more than they help. To help SPASS cope better with large theories, we experimented with two custom strategies.

The *goal-based* strategy first chooses the negated conjecture and each axiom in turn from the usable set; this way, single-inference proofs are found early if they exist (i.e., if the conjecture is implied by an axiom). From then on, only clauses employing allowed symbols may be selected. Initially, the set of allowed symbols consists of those appearing in the conjecture. If no appropriate clause is available, the strategy looks for inferences that produce such a clause; failing that, the lightest clause is chosen, its symbols are added to the allowed symbols, and the maximal depth is incremented.

The *ranks-based* strategy requires each clause to carry a *rank* indicating its likely relevance. Like the goal-based strategy, it first selects the negated conjecture and the axioms. From then on, it always chooses the clause that minimizes the product *weight* $\times$ *depth* $\times$ *rank*. The rank of a derived clause is the minimum of the ranks of its parents. Conveniently, the facts returned by Sledgehammer's relevance filter are ordered by syntactic closeness to the goal to prove, a rough measure of relevance. The formula for assigning ranks interpolates linearly between .25 (for the first fact) and 1 (for the last fact). We have yet to experiment with other coefficients and interpolation methods.

Having several strategies to choose from may seem a luxury, but in conjunction with a simple optimization, *time slicing*, it helps find more proofs, especially if the strategies complement each other well. Automatic provers rarely find proofs after having run unsuccessfully for a few seconds, so it usually pays off to schedule a sequence of strategies, each with a fraction (or slice) of the total time limit. This idea can be extended to other aspects of the translation and proof search: the number of facts, type encodings, and λ-abstraction translation schemes, as well as various prover options.

Sledgehammer implements time slicing for any automatic prover, so that it is run with its own optimized schedule. We used a greedy algorithm to compute a schedule for SPASS based on a standard benchmark suite (Judgment Day [10]). The schedule incorporates both the goal- and the rank-based strategies, sometimes together with SOS.

6 Case Study: Formalization of Language-Based Security

As part of a global trend toward formalized computer science, the interactive theorem proving community has in recent years become interested in formalizing aspects of language-based security [37]. The pen-and-paper proofs of security results (typically, the soundness of a security type system) tend to be rather involved, requiring case analyses on the operational semantics rules and tedious bisimilarity reasoning. Formalizations of these results remain rare heroic efforts.

To assist such efforts, we are developing a framework to reason uniformly about a variety of security type systems and syntax-directed quantitative analyses. The central notion is that of *compositional bisimilarity relations*, which yield sound type systems that enforce security properties. The bulk of the development establishes compositionality of language constructs (e.g., sequential and parallel composition, 'while') with respect to bisimilarity relations (e.g., strong, weak, 01-bisimilarity) [33].

We rely heavily on Sledgehammer to automate the tedious details. Besides easing the proof development, automation helps keep the formal proofs in close correspondence with the original pen-and-paper proof. To illustrate this point, we review a typical proof of compositionality: sequential composition (;) with respect to strong bisimilarity ($\approx$). The goal is $c_1 \approx d_1 \wedge c_2 \approx d_2 \Longrightarrow c_1;c_2 \approx d_1;d_2$. The proof involves three steps:

1. Define a relation R that witnesses bisimilarity of $c_1;c_2$ and $d_1;d_2$.
2. Show that R is a bisimulation.
3. Conclude that $R \subseteq\, \approx$ by the definition of $\approx$ as greatest bisimulation.

Step 1 is the creative part of the argument. Here, the witness is unusually straightforward: $R = \{(c,d).\ \exists c_1\, c_2\, d_1\, d_2.\ c = c_1;c_2\ \wedge\ d = d_1;d_2\ \wedge\ c_1 \approx d_1\ \wedge\ c_2 \approx d_2\}$. Steps 2 and 3 are left to the reader in textbook presentations and constitute good candidates for full automation. Unfortunately, the goal is beyond the reach of Isabelle's proof methods, and the translated first-order goals are too difficult for the automatic provers.

For step 2, we must show that if $(c,d) \in R$ and s and t are states indistinguishable to the attacker, written $s \sim t$, then any step taken by c in state s is matched by a step taken by d in state t such that the continuations are again in R and the resulting states are again indistinguishable. Assume $c = c_1;c_2$ and $(c,s) \to (c',s')$ represents a step taken by c in state s with continuation c' and resulting state s'.

2.1. By rule inversion for the semantics of sequential composition, either
   a. c' has the form $c_1';c_2'$ and $(c_1,s) \to (c_1',s')$; or
   b. $(c_1,s) \to s'$ (i.e., c_1 takes in s a terminating step to s') and $c' = c_2$.

2.2. Assume case a. From $c_1 \approx d_1$ we obtain d_1' and t' such that $(d_1,t) \to (d_1',t')$, $c_1' \approx d_1'$, and $s' \sim t'$. (Case b is analogous, with $c_2 \approx d_2$ instead of $c_1 \approx d_1$.)

2.3. By one of the *intro* rules for the semantics of sequential composition, namely, $(c_1,s) \to (c_1',s') \Longrightarrow (c_1;c_2,s) \to (c_1';c_2,s')$, we obtain $(d_1;d_2,t) \to (d_1';d_2,t')$. Hence, the desired matching step for $d = d_1;d_2$ is $(d,t) \to (d_1';d_2,t')$.

Until recently, we would discharge 2.2 and 2.3 by invoking the simplifier enriched with the *intro* rules for the language construct (here, sequential composition) followed by Sledgehammer. The SPASS integration now allows us to discharge 2.2 and 2.3 without the need to customize and invoke the simplifier. In addition, if we are ready to wait a full minute, SPASS can discharge the entire step 2 in a single invocation, replacing the old cliché "left to the reader" with the more satisfying "left to the machine."

SPASS also eases reasoning about execution traces, modeled as lazy lists. Isabelle's library of lazy list lemmas is nowhere as comprehensive as that of finite lists. Reasoning about lazy lists can often exploit equations relating finite and lazy list operations via coercions. For example, under the assumption that the lazy lists xs and ys are finite, the following equations push the list_of coercion inside terms:

$$\mathsf{list\_of}\ (\mathsf{LCons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{list\_of}\ xs)$$
$$\mathsf{list\_of}\ (\mathsf{lappend}\ xs\ ys) = \mathsf{append}\ (\mathsf{list\_of}\ xs)\ (\mathsf{list\_of}\ ys)$$

The proper orientation of such equations helps discharge many goals for which we previously needed to engage at a tiresome level of detail.

7 Evaluation

This section attempts to quantify the enhancements described in Sections 3 to 5, both by evaluating each new feature in isolation and by letting the new SPASS compete against other automatic provers.[3] Our benchmarks are partitioned into three sets:

- *Judgment Day* (JD, 1268 goals) consists of seven theories from the Isabelle distribution and the *Archive of Formal Proofs* [21] that served as the main benchmark suite for Sledgehammer over the last two years [5,6,10,31]. It covers areas as diverse as the fundamental theorem of algebra, the completeness of a Hoare logic, and the type soundness of a Java subset.
- *Arithmetic Extension of Judgment Day* (AX, 616 goals) consists of three Isabelle theories involving both linear and nonlinear arithmetic that were used in evaluations of SMT solvers and type encodings [5,6].
- *Language-Based Security* (LS, 1042 goals) consists of five Isabelle theories belonging to the development described in Section 6.

Running Sledgehammer on a theory means launching it on the first subgoal at each position where a proof command appears in the theory's script.

To test the new SPASS features, we defined a base configuration and test variants of the configuration where one feature is disabled or replaced by another. Hard sorts, advisory and mandatory simplification, and goal-based clause selection are all part of the base configuration. The generated problems include up to 500 facts (700 after monomorphization). For each problem, SPASS is given 60 seconds of one thread on 64-bit Linux systems equipped with two Quad-Core Intel Xeon processors running at 2.4 GHz. We are interested in proof search alone, excluding proof reconstruction.

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. The benchmarks were chosen to be representative of typical Isabelle goals and include many that are either too easy or too hard to effectively evaluate automatic provers. Indeed, some of the most essential tools in Isabelle, such the arithmetic decision procedures, score well below 10% when applied indiscriminately to the entire Judgment Day suite. Furthermore, SPASS has been fine-tuned over nearly two decades; it would be naive to expect enormous gains from isolated enhancements.

With this caveat in mind, let us review Fig. 1. It considers six representations of types: three polymorphic and three monomorphic. Guards and tags are two traditional encodings. "Type arguments" was until recently the default in Sledgehammer; its actual success rate after reconstruction is lower than indicated, because some of the proofs found with it are unsound. "Light guards" is the new lightweight guard-based encoding. Many other encodings are implemented; in terms of performance, they are sandwiched between polymorphic guards and monomorphic light guards [6]. The results are fairly consistent across benchmark sets and confirm that hard sorts are superior to any encoding. The better translation schemes are also noticeably faster: Proofs with hard sorts require 3.0 seconds on average, compared with 5.0 for monomorphic guards.

[3] The test data set is available at `http://www21.in.tum.de/~blanchet/itp2012-data.tgz`

* unsound	JD	AX	LS	All
Guards	28.7	17.9	29.7	25.6
Tags	37.3	23.8	39.1	33.5
Type args.*	**46.9**	**30.8**	**51.6**	**42.6**

(a) Polymorphic

	JD	AX	LS	All
Guards	40.7	29.8	46.1	37.9
Light guards	50.5	33.5	50.0	45.6
Hard sorts	**52.0**	**34.1**	**50.8**	**46.7**

(b) Monomorphic

Fig. 1. Success rates (%) for the main type encodings

	JD	AX	LS	All
Advisory	+2.7	−2.9	+0.4	+0.9
Mandatory	+3.2	−**1.4**	+1.5	+1.8
Both	+**3.8**	−1.9	+**1.9**	+**2.2**

(a) Simplification

	JD	AX	LS	All
SOS	+9.9	−10.4	−4.2	+1.1
Goal-based	+15.3	+2.0	−1.5	+6.5
Rank-based	+**17.6**	+**6.9**	+**9.2**	+**12.6**

(b) Clause selection

Fig. 2. Improvements (%) over the default setup for each proof heuristic

Fig. 2(a) presents the impact of advisory and mandatory simplification as a percentage improvement over a configuration with both features disabled. The overall gain is 2.2% (i.e., SPASS solves 102.2 goals with both mechanisms enabled whenever it would solve 100 goals without them). Fig. 2(b) compares three clause selection strategies with SPASS's default strategy. Our custom goal- and rank-based strategies are considerably more successful than the traditional SOS approach.

Fig. 3 shows how the main clause-selection strategies scale when passed more facts, compared with the default setting. Both custom strategies degrade much less sharply than the default. The goal-based strategy scales the best, with a peak at 800 facts compared with 150 for the default strategy and 400 for the rank-based strategy. There is potential for improvement: With rank annotations, it should be possible to design a strategy that actually improves with the number of facts. As a thought experiment, a variant of the goal-based strategy that simply ignores all facts beyond the 800th would flatly outclass every strategy on Fig. 3 when given, say, 900 or 1000 facts.

Finally, we measure the new version of SPASS against the latest versions of E (1.4), SPASS (3.7), Vampire (1.8 rev. 1435), and Z3 (3.2). Vampire and Z3 support hard sorts, and Z3 implements arithmetic decision procedures. This evaluation was conducted on the same hardware as the original Judgment Day study: 32-bit Linux systems with a Dual-Core Intel Xeon processor running at 3.06 GHz. The time limit is 60 seconds for proof search, potentially followed by minimization and reconstruction.

Fig. 4(a) gives the success rate for each prover on each benchmark set. Overall, SPASS now solves 13% more goals than before, or 6.1 percentage points; this is enough to make it the single most useful automatic prover. About 45% of the goals from the chosen Isabelle theories are "trivial" in the sense that they can be solved directly by standard proof methods invoked with no arguments. If we ignore these and focus on the more interesting 1625 nontrivial goals, SPASS's improvements are even more significant: It solves 21% more goals than before (corresponding to 6.5 percentage points) and 10% more than the closest competitor (3.3 percentage points), as shown in Fig. 4(b).

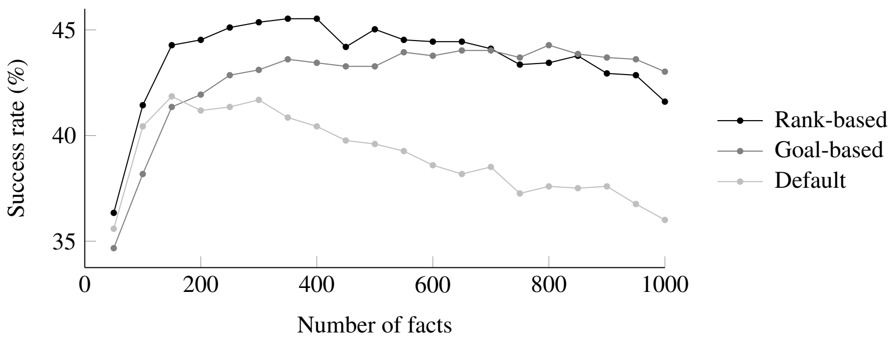


Fig. 3. Scalability of each clause-selection strategy

	JD	AX	LS	All
New SPASS	**56.2**	38.3	**60.4**	**53.9**
Z3	53.5	**40.4**	57.6	52.2
Vampire	54.2	35.2	57.9	51.5
E	53.0	39.0	56.1	51.2
Old SPASS	50.0	31.8	54.6	47.8
Together	63.6	51.3	67.8	62.5

(a) All goals

	JD	AX	LS	All
New SPASS	**41.4**	**23.7**	**41.3**	**37.4**
Z3	38.6	21.3	36.7	34.1
Vampire	38.7	19.9	37.4	34.0
E	37.0	22.4	38.2	34.0
Old SPASS	34.2	19.7	34.2	30.9
Together	49.3	32.6	46.6	44.7

(b) Nontrivial goals

Fig. 4. Success rates (%) with proof reconstruction for selected provers

Two years ago, the combination of (older versions of) E, SPASS, and Vampire running in parallel for 120 seconds solved 48% of Judgment Day [10]. Largely thanks to the developments presented in this paper, SPASS alone now solves 56% of the benchmark suite in half of the time, on the very same hardware.

8 Related Work

The most notable integrations of automatic provers in proof assistants, either as oracles or with proof reconstruction, are probably Otter in ACL2 [25]; Bliksem and veriT in Coq [2,4]; Gandalf in HOL98 [18]; Z3 in HOL4 [11]; CVC Lite in HOL Light [26]; Vampire in Mizar [35]; Bliksem, EQP, LEO, Otter, PROTEIN, SPASS, TPS, and Waldmeister in ΩMEGA [39]; and Yices in PVS [36]. Of these, only LEO and Yices appear to have been significantly tailored to their host system. For program verification, Z3 in Boogie [3] and Alt-Ergo in Why3 [8] are examples of integrated proof environments.

Much of the developments currently taking place in first-order automatic theorem provers focus on solving particular classes of problems. This includes, for example, the automatic generation of inductive invariants for some theory or the efficient decision of large ontologies belonging to some decidable first-order fragment. From this point of view, our work on tailoring SPASS toward a better combination with Isabelle is the first dedicated contribution of its kind.

9 Conclusion

This paper described a tight, dedicated integration of Isabelle and SPASS. The heart of the approach is to communicate in a rich format that augments classical first-order logic with annotations for sorts, simplification rules, and fact relevance, and to let that information guide the proof search. The new version of SPASS outperforms E, Vampire, and Z3 on our extensive benchmark suites, and it is already helping us fill in the tedious details in language-based security proofs.

There is much room for future work, notably to support polymorphism and to extend the configurable simplification mechanisms to inductive and coinductive predicates and their *intro* and *elim* rules. It would also be desirable to polish and exploit SPASS's hierarchical support for linear and nonlinear arithmetic [1,13] and accommodate additional theories, such as algebraic datatypes, that are ubiquitous in formal proof developments. Finally, a promising avenue of work that could help derive deeper proofs within the short time allotted by Sledgehammer would be to have SPASS cache inferences across invocations, instead of re-deriving the same consequences from the same background theories over and over again.

We hope this research will serve as a blueprint for others to tailor their provers for proof assistants. Interactive theorem proving provides at least as many challenges as the annual competitions that are so popular in the automated reasoning community. Granted, there are no trophies or prizes attached to these challenges (a notable exception being the ISA category at CASC-23 [41]), but the satisfaction of assisting formalization efforts should be its own reward.

Acknowledgment. We thank Tobias Nipkow for making this collaboration possible and discussing the draft of this paper. We are also grateful to Lukas Bulwahn, Laura Faust, Lawrence Paulson, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements. This research was supported by the project Security Type Systems and Deduction (grants Ni 491/13-1 and We 4473/1-1) as part of the program Reliably Secure Software Systems (RS^3, Priority Program 1496) of the Deutsche Forschungsgemeinschaft (DFG). The first author was supported by the DFG project Quis Custodiet (grant Ni 491/11-2).

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition Modulo Linear Arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer, Heidelberg (2009)
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. 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)

4. Bezem, M., Hendriks, D., de Nivelle, H.: Automatic proof construction in type theory using resolution. J. Autom. Reas. 29(3-4), 253–275 (2002)
5. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT Solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 116–130. Springer, Heidelberg (2011)
6. Blanchette, J.C., Böhme, S., Smallbone, N.: Monotonicity or how to encode polymorphic types safely and efficiently, http://www21.in.tum.de/~blanchet/mono-trans.pdf
7. Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism, http://www21.in.tum.de/~blanchet/tff1spec.pdf
8. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT 2008, pp. 1–5. ICPS, ACM (2008)
9. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 87–102. Springer, Heidelberg (2011)
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. Böhme, S., Weber, T.: Fast LCF-Style Proof Reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010)
12. 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)
13. Eggers, A., Kruglov, E., Kupferschmid, S., Scheibler, K., Teige, T., Weidenbach, C.: Superposition Modulo Non-linear Arithmetic. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 119–134. Springer, Heidelberg (2011)
14. Filliâtre, J.C.: Private Communication (March 2012)
15. Ganzinger, H., Meyer, C., Weidenbach, C.: Soft Typing for Ordered Resolution. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 321–335. Springer, Heidelberg (1997)
16. Gonthier, G.: Formal proof—The four-color theorem. N. AMS 55(11), 1382–1393 (2008)
17. 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)
18. Hurd, J.: Integrating Gandalf and HOL. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 311–321. Springer, Heidelberg (1999)
19. 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, No. CP-2003-212448 in NASA Technical Reports (2003)
20. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. C. ACM 53(6), 107–115 (2010)
21. Klein, G., Nipkow, T., Paulson, L. (eds.): The Archive of Formal Proofs, http://afp.sf.net/
22. Knuth, D.E., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
23. Leroy, X.: A formally verified compiler back-end. J. Autom. Reas. 43(4), 363–446 (2009)
24. Leroy, X.: Private Communication (October 2011)
25. McCune, W., Shumsky, O.: System Description: IVY. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 401–405. Springer, Heidelberg (2000)
26. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. ENTCS 144(2), 43–51 (2006)

27. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reas. 40(1), 35–60 (2008)
28. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
29. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
30. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
31. 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)
32. Paulson, L.C., Susanto, K.W.: Source-Level Proof Reconstruction for Interactive Theorem Proving. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 232–245. Springer, Heidelberg (2007)
33. Popescu, A., Hölzl, J., Nipkow, T.: Proving possibilistic, probabilistic noninterference, http://www21.in.tum.de/~popescua/pos.pdf (submitted)
34. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. 15(2-3), 91–110 (2002)
35. Rudnicki, P., Urban, J.: Escape to ATP for Mizar. In: Fontaine, P., Stump, A. (eds.) PxTP 2011 (2011)
36. Rushby, J.M.: Tutorial: Automated formal methods with PVS, SAL, and Yices. In: Hung, D.V., Pandya, P. (eds.) SEFM 2006, p. 262. IEEE (2006)
37. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Comm. 21(1), 5–19 (2003)
38. 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)
39. Siekmann, J., Benzmüller, C., Fiedler, A., Meier, A., Normann, I., Pollet, M.: Proof development with ΩMEGA: The irrationality of $\sqrt{2}$. In: Kamareddine, F. (ed.) Thirty Five Years of Automating Mathematics. Applied Logic, vol. 28, pp. 271–314. Springer, Heidelberg (2003)
40. Sutcliffe, G.: The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. J. Autom. Reas. 43(4), 337–362 (2009)
41. Sutcliffe, G.: The CADE-23 automated theorem proving system competition—CASC-23. AI Comm. 25(1), 49–63 (2012)
42. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP Typed First-Order Form with Arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 406–419. Springer, Heidelberg (2012)
43. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, pp. 1965–2013. Elsevier (2001)
44. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009)
45. Wos, L., Robinson, G.A., Carson, D.F.: Efficiency and completeness of the set of support strategy in theorem proving. J. ACM 12(4), 536–541 (1965)

A Language of Patterns for Subterm Selection[⋆]

Georges Gonthier[1,2] and Enrico Tassi[2,3]

[1] Microsoft Research Cambridge
[2] INRIA - Microsoft Research Joint Centre
[3] INRIA, Laboratoire d'Informatique de l'Ecole Polytechnique
gonthier@microsoft.com, enrico.tassi@inria.fr

Abstract. This paper describes the language of patterns that equips the SSReflect proof shell extension for the Coq system. Patterns are used to focus proof commands on subexpressions of the conjecture under analysis in a declarative manner. They are designed to ease the writing of proof scripts and to increase their readability and maintainability.

A pattern can identify the subexpression of interest approximating the subexpression itself, or its enclosing context or both. The user is free to choose the most convenient option.
Patterns are matched following an extremely precise and predictable discipline, that is carefully designed to admit an efficient implementation.

In this paper we report on the language of patterns, its matching algorithm and its usage in the formal library developed by the Mathematical Components team to support the verification of the Odd Order Theorem.

1 Introduction

In the design of proof languages many aspects have to be considered. Among them, the one that interests us the most is efficiency, both in writing proof scripts and in fixing them when they break.

Efficiency in writing and maintaining scripts is a crucial aspect for a language to be successfully adopted in a large development like the Mathematical Components library. That library comprises more than ten thousands lemmas, spread over one hundred files totalling over 113 thousand lines of code. For this development the SSReflect proof language was chosen, after its successful use in the formalization of the Four Color Theorem [10].

SSReflect is an extension of the Coq system, and inherits some of its strengths and weaknesses. The higher order logic of Coq allows to use computation as a form of proof and to enforce many invariants through its rich type system. These features were key ingredients in the formalization of Four Color Theorem, as well as in the development of many decision procedures [6,11], in interfacing Coq with external tools [18,2] and in organizing mathematical theories into higher-level packages [8,16]. The down side of this sophistication is that all basic operations, such as

⋆ The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

term comparison or term matching, have to take computation into account, thus becoming harder to predict due to the additional complexity.

SSReflect achieves *efficiency in writing* proof scripts giving the user a very precise and predictable language to assemble together carefully stated lemmas. The language is designed to be compact and compositional, with very few basic building blocks. Among them rewriting plays a special role and is indeed the most often used proof command. We describe how we improved its expressiveness while retaining, and in some circumstances even improving, its predictability.

To achieve *efficiency in maintaining* scripts the language constructs are equipped with a precise semantics that forces failures to happen early and locally. This is supplemented by support for script readability. SSReflect encourages to mix declarative steps, which assert intermediate results, and procedural statements that prove them. Declared statements are check points from which the user is likely to start replaying a broken proof, and the closer the failure is to one of these check points the easier is the fix. In this paper we describe how the `rewrite` command was made more declarative and less ambiguous.

Rewriting in particular, but also any other command that deals with subexpressions of the current conjecture, can in fact be rather ambiguous. Similar subexpressions are quite common in large conjectures and rewrite rules usually admit several different instantiations, each of which may occur multiple times. The two sources of ambiguity are thus: 1) the *instantiation* of the rewriting rule arguments to obtain a completely specified expression; 2) the *selection of the occurrences* of this expression to be affected. The standard approach to cope with the first source of ambiguity is to manually instantiate the rewriting rule. This approach requires the user to remember the nature and the order or names of the arguments of any rule, and thus hardly scales to a library with thousands of rewriting rules. Occurrences are usually selected by numbers. As we will show, this turns out to be rather inconvenient for script maintenance.

In this paper we describe the different approach adopted in the SSReflect proof language. The user is given a language of patterns to express in a concise and precise way which subterms of the current conjecture are affected by proof commands like `rewrite`.

The SSReflect Coq extension version 1.3pl2 for Coq 8.3 is available for download at the Mathematical Components web site[1]. The reader is not required to be acquainted with the specific logic of Coq or the proof language of SSReflect, but may find the reference manuals of the two tools [13,9] helpful.

In Section 2 we describe the language of patterns and give some examples on their intended use. Section 3 details the term matching algorithm. Section 4 compares our approach with the ones adopted in Coq, Matita and Isabelle/Isar.

2 A Language of Patterns

Most lines in a SSReflect proof script are procedural: they modify the current conjecture without explicitly stating the expected result. The `rewrite` command, whose argument is a list of rules, is a perfect example of this: instead of displaying a list of conjectures and expecting the system (and reader) to guess how to change one line into the next, the user quotes explicitly the *rules names* that justify the changes, and lets Coq do them sequentially. As Coq can usually reliably figure out how to apply each rule, this avoids a repetition of the parts of the conjecture that are *not* concerned by each change, and this is good for both writing and maintaining.

For example, consider these inequalities taken from Theorem 14.7 [4], that is the main result of the Local Analysis part of the Odd Order Theorem:

$$g \leq \left(1 + \frac{n}{z} - \sum_{M_i \in MX} (k_{M_i})^{-1} + \sum_{M_i \in MX} ((k_{M_i})^{-1} - (2z)^{-1})\right) \cdot g$$

$$g \leq \left(1 + \frac{n}{z} - \sum_{M_i \in MX} (k_{M_i})^{-1} + \sum_{M_i \in MX} (k_{M_i})^{-1} + (-(2z)^{-1} \cdot |MX|)\right) \cdot g$$

Rather than spelling out the second term above, we explicitly describe how to turn the fist inequality into the second one: the rightmost summation is split into two simpler ones using `big_split`; then the resulting summation of constant terms $\sum_{M_i \in MX}(-(2z)^{-1})$ is solved with `sumr_const`. This manipulation is expressed by the following command:

```
rewrite big_split sumr_const.
```

This is clearer and much more concise than the complex term it yields; indeed, in the actual proof we add more rules to carry out further simplifications.

However, sometimes guidance *is* needed, and we claim that it is best provided declaratively, using a little pattern language. Patterns are declarative in the sense that the user writes (an approximation of) the subterm that should be affected by a proof command. We will however assign a precise procedural interpretation to the pattern language in order to preserve the predictability of proof commands. It is also worth mentioning that the subterm a proof command may be focused on is often much smaller than the whole conjecture, and can usually be approximated by an even smaller pattern. So compactness is also retained.

We now give an informal presentation of patterns by some examples. We show how ambiguities in the execution of the `rewrite` proof command can be avoided thanks to a pattern and why the solution we propose is superior to existing ones.

Example 1 (Simple pattern)

This example lies in the context of the algebraic hierarchy [8] of the SSReflect library, where the infix notation for subtraction is a short hand for the addition of the opposite.

```
Infix "a - b" = (a + -b).
Lemma addrC x y : x + y = y + x.
Lemma addNKr x y : x + (- x + y) = y.

Lemma example : a + (b * c - a) + a = b * c + a.
```

The idea in the proof that will follow is to cancel the leftmost `a` with its opposite `-a`, thanks to `addNKr`. To rewrite with that lemma a preliminary step to move `-a` closer to `a` is needed: commutativity of addition must be used *on the correct subexpression.* Unluckily there are many occurrences of the addition operation. If no extra information is provided, the left hand side of the rule is the very ambiguous pattern `(_ + _)`, where `_` denotes a wild card. Its first instance encountered in pre-visit order is `(a + (b * c - a) + a)` that is not not the desired one, so additional information to disambiguate the rule is really needed.

To cope with this first form of ambiguity, *instantiation*, we have to specify the values of the quantified variables `x` and `y` to `addrC`. The standard approach adopted in many procedural proof languages, like the Coq's standard one, is to instantiate these variables manually, as in one of the following commands:

```
rewrite (addrC (b * c) (-a))          rewrite (addrC (x := b * c))
```

Both the previous commands turn the conjecture in the following one. From now on we will use a wave to underline the effect of a proof command.

$\mathtt{a + \underset{\sim\sim\sim\sim\sim\sim\sim\sim}{(-a + b * c)} + a = b * c + a}$

In the first case `x` and `y` are passed as arguments to the lemma by position. The left hand side of the rule becomes `(b * c - a)`. This expression has just one instance in the conjecture, thus the second kind of ambiguity, *occurrence selection*, does not occur. The second command passes the argument for `x` by name and leaves `y` undefined. The left hand side of the rule is thus a pattern `(b * c + _)` where `_` is a wild card. The system looks for an instance of that pattern in a prefix traversal of the conjecture, again finding the correct instance.

As anticipated in the introduction the main problem of this approach is that the user has to remember the order in which the variables of a rewrite rule are abstracted, or their names. What looks easy for the simple common lemma `addrC` quickly becomes an issue in the context of a large formalization like the one for the Odd Order Theorem, comprising over ten thousands lemmas.

The approach we propose is not only solving this usability issue but is also more compact, as shown in the following snippet.

```
rewrite [_ - a]addrC
```

The square brackets prefixing the rewrite rule `addrC` delimit the pattern `(_ - a)`. The pattern has a single, non ambiguous instance in the conjecture, namely `(b * c - a)`. Prefixing the rewriting rule name with a pattern the user substitutes the inferred pattern with a more specific one, better approximating the instance on which she wants to focus the proof command. □

A good interface design is most crucial to the usability of a theory library, and achieving one often requires several rounds of incremental refinement. When a potential improvement is identified, the statement of many lemmas is changed accordingly and their proofs are likely to break and thus require time consuming maintenance work. The general approach of the SSReflect language to lowering the cost of these refactoring activities is to detect failures as early as possible.

Example 2 (Proof script breakage). The lemma of the previous statement could be replaced (on purpose or by accident) by the following one:

$$\texttt{a + (b * c }\uwave{\texttt{+ a}}\texttt{) + a = b * c + a.}$$

The user provided pattern `[_ - a]` seen before has no instance in this conjecture thus failure is immediately detected. On the contrary the command where `x` is instantiated by name with `(b * c)` would continue to produce an output, even if a different one. In that case the pattern `(b * c + _)` does have an instance occurring twice in the conjecture, namely `(b * c + a)`. Instead of signalling an error, the system changes the conjecture into the following one.

$$\texttt{a + }\uwave{\texttt{(a + b * c)}}\texttt{ + a = }\uwave{\texttt{a + b * c}}\texttt{.}$$

Failure will then happen at a later stage, with a conjecture that is very different from the one the author of the original proof script was seeing. Moreover the original intention of the user to move `(-a)` to the left can be recognized in the pattern `[_ - a]` but not in the instantiation `(addrC (x := b * c))`. □

As mentioned in the introduction the logic of Coq identifies terms up to conversion, i.e., unfolding of definitions and recursive functions computation. To develop a large library in a convenient way, the user often defines new concepts in terms of preexisting ones. In most cases part of the theory already developed naturally transports to the new concepts. As we see in the following example this may introduce an additional degree of ambiguity the user has to deal with.

Example 3 (Pattern forcing definition unfolding). In the context of the library on lists the user finds the function `map` to apply a function over a list and some of its properties. The related lemma `eq_in_map` states that a function `f` can be replaced with another function `g` if `f` and `g` are point wise equal (denoted `=1`) on the list they are mapped on. `map_comp` proves that mapping two functions in a row is the same as mapping their functional composition (denoted with `\o`). `id` is the identify function.

```
Lemma eq_in_map s f g : {in s, f =1 g} -> map f s = map g s.
Lemma map_comp f g s : map (f \o g) s = map f (map g s).
Lemma map_id s : map id s = s.
```

The `iota` function builds the list of consecutive integers given the first element and the list length. On top of `map` and `iota` the user defines the `graph` of a function over an integer interval `[0,n[` as the list of its values on that interval. An obvious property is that if a function `f` behaves as the identity on the graph of `g` on a given interval, then the graph of `(f \o g)` is equal to the graph of `g` on the same interval.

```
Definition graph f n := map f (iota 0 n).
Lemma graph_comp f g n (pf : {in graph g n, f =1 id}) :
  graph (f \o g) n = graph g n.
```

The property follows trivially from the theory of lists, but the conjecture does not mention any list operation. Nevertheless the `map_comp` rewrite rule can be used as follows:

```
rewrite [graph _ n]map_comp
```

The first instance of the pattern `[graph _ n]`, traversing the conjecture `(graph (f \o g) n = graph g n)` from left to right, is `(graph (f \o g) n)`. This is where the `map_comp` rule can apply. In fact, unfolding the definition of `graph` exposes `(map (f \o g) (iota 0 n))` that is clearly an instance of the pattern `(map (_ \o _) _)` given by the rule `map_comp`. The resulting conjecture is reported below.

```
map f (map g (iota 0 n)) = graph g n.
```

One can then complete the proof rewriting with the `eq_in_map` lemma, whose hypothesis is indeed equivalent to the `pf` assumption, and then conclude with `map_id`. □

One could argue that in the previous example the system is not "clever enough" and could exploit the fact that `graph` is defined in terms of `map` to find the subterms to be rewritten. According to our experience this would make the rewrite command less predictable. For example consider a conjecture in which both `graph` and `map` occur in that order. The `graph` occurrence may be rewritten even if the user does not know that graph is defined in terms of `map`. Moreover the user still needs a way to focus on `map` if that is what she wants.

The usual alternative approach is to manually unfold some of the occurrences of `graph` to expose `map`. This is again not only more verbose, but less informative in the script. With the pattern `[graph _ n]` the user clearly states that the whole matched expression is an instance of the left hand side of the rewriting rule. If `graph` is redefined with a different expression that strictly contains an occurrence of `map`, the script with the pattern breaks immediately, while the one just unfolding `graph` may signal an error at a later stage.

The previous examples may look a bit artificial, and in fact they were chosen to be reasonably self contained at the cost of resulting a bit simplistic. On the contrary the one below is taken from the quite involved proof of the Wielandt fixpoint [12] Theorem formalized by A. Mahboubi. It is a rather technical result required to prove the Odd Order Theorem, and was one of the motivating examples for contextual patterns, that are introduced immediately after.

Example 4 (Contextual pattern). The context of this example is group theory and the study of group morphisms. The system prints above the double bar the hypotheses accumulated by the user so far. In particular that `X` is equal to the image of the morphism `fact_g` over `X` quotiented by the kernel of `g`. The user needs to rewrite the first occurrence of `X` with the `imgX` equation in order to advance in her proof.

```
nkA : joing_group A X \subset 'N('ker g)
fact_g := factm skk nkA : coset_groupType ('ker g) -> gT
imgX : X = fact_g @* (X / 'ker g)
=================================
minnormal (fact_g @* (A / 'ker g)) X ->
  minnormal (A / 'ker g) (X / 'ker g)
```

Here the rewrite rule is fully instantiated, thus the ambiguity is given by the fact that its left hand side `X` occurs at least twice in the conjecture (the implication below the double bar). In fact, the notation system of COQ hides many other occurrences of `X`. The morphism image construction `@*` is polymorphic over the type of the morphism `fact_g`, that is itself a dependently typed construction. In particular it depends on the assumption `nkA` whose type mentions `X`. The logic of COQ features explicit polymorphism, like system $\mathcal{F}$, so types occur as arguments of polymorphic functions even if some syntactic sugar hides them. As it turns out, the occurrence of `X` we are interested in is number twenty-nine, even if it the first one displayed by the system.

The pattern we propose to unambiguously identify that desired occurrence uses its enclosing context. In the following snippet, `R` is a name bound in the expression following the `in` keyword.

```
rewrite [R in minnormal _ R]imgX
```

The intended meaning is to focus the `rewrite` command on the subterm identified by `R` in the first occurrence of the context `(minnormal _ R)`. While being more verbose than the occurrence number `{29}`, it is way easier to write, since no guessing is needed. Moreover in case the script breaks the original intent of the user is clearly spelled out. □

2.1 Syntax and Semantics

The syntax is defined by two grammar entries: ⟨*c-pattern*⟩ for contextual patterns and ⟨*r-pattern*⟩ for their superset rewrite patterns. Contextual patterns are meant to identify a specific subterm, and can be used as arguments of the SSREFLECT commands `set`, `elim` and `:` (colon), see [9, Sections 4.2 and 5.3], respectively used to declare abbreviations, perform induction or generalize the current conjecture. Rewrite patterns are a strict superset of contextual patterns adding the possibility of identifying all the subterms under a given context. They can be used as arguments of the SSREFLECT `rewrite` command, see [9, Section 7].

$$\langle \textit{c-pattern} \rangle ::= [\langle \textit{tpat} \rangle\ \texttt{as} \mid \langle \textit{tpat} \rangle\ \texttt{in}]\ \langle \textit{ident} \rangle\ \texttt{in}\ \langle \textit{tpat} \rangle \mid \langle \textit{tpat} \rangle$$
$$\langle \textit{r-pattern} \rangle ::= \langle \textit{c-pattern} \rangle \mid \texttt{in}\ [\langle \textit{ident} \rangle\ \texttt{in}]\ \langle \textit{tpat} \rangle$$

Here ⟨*tpat*⟩ denotes a term pattern, that is a generic COQ term, possibly containing wild cards, denoted with `_` (underscore).

We now summarize the semantics of both categories of patterns. We shall call *redex* the pattern designed to identify the subterm on which the proof command will have effect. We shall also use the word *match* in an informal way recalling the reader's intuition to pattern matching. The precise meaning of matching will be described in Section 3.

Contextual Patterns ⟨*c-pattern*⟩. For every possible pattern we identify the *redex* and define which subterms are affected by a proof command that uses such pattern. We then point out the main subtleties with some examples.

$\langle$***tpat***$\rangle$ is the degenerate form of a contextual pattern, where the context is indeed empty. The redex is thus the whole $\langle$*tpat*$\rangle$. The subterms affected by this simple form of pattern are all the occurrences of the first instance of the redex. See Example 5.

$\langle$***ident***$\rangle$ **in** $\langle$***tpat***$\rangle$ is the simplest form of contextual pattern. The redex is the subterm of the context $\langle$*tpat*$\rangle$ bound by $\langle$*ident*$\rangle$. The subterm affected are all the subterms identified by the redex $\langle$*ident*$\rangle$ in all the occurrences of the first instance of $\langle$*tpat*$\rangle$. See Example 6.

$\langle$***tpat***$\rangle_1$ **as** $\langle$***ident***$\rangle$ **in** $\langle$***tpat***$\rangle_2$ is a form of contextual pattern where the redex is explicitly given as $\langle$*tpat*$\rangle_1$. It refines the previous pattern by specifying a pattern for the context hole named by $\langle$*ident*$\rangle$. The subterms affected are thus the ones bound by $\langle$*ident*$\rangle$ in all the occurrences of the first instance of $\langle$*tpat*$\rangle_2[\langle$*tpat*$\rangle_1/\langle$*ident*$\rangle]$, i.e., $\langle$*tpat*$\rangle_2$ where $\langle$*ident*$\rangle$ is replaced by $\langle$*tpat*$\rangle_1$. See Example 8.

$\langle$***tpat***$\rangle_1$ **in** $\langle$***ident***$\rangle$ **in** $\langle$***tpat***$\rangle_2$ is the last form of contextual pattern and is meant to identify deeper contexts in two steps. The redex is given as $\langle$*tpat*$\rangle_1$ and the subterms affected are all the occurrences of its first instance inside the subterms bound by $\langle$*ident*$\rangle$ in all the occurrences of the first instance of $\langle$*tpat*$\rangle_2$. The context described by this pattern is thus made of two parts: an explicit one given by $\langle$*tpat*$\rangle_2$, and an implicit one given by the matching of the redex $\langle$*tpat*$\rangle_1$ that could occur deep inside the term identified by $\langle$*ident*$\rangle$. See Example 7.

Example 5. We have already seen in Example 4 the first form of pattern. Here we give another example to stress that *all* the occurrences of the first instance of the pattern are affected. Take the conjecture:

```
(a - b) + (b - c) = (a - b) + (d - b)
```

The proof command `rewrite [_ - b]addrC` changes the conjecture as follows because the first instance of the pattern `(_ - b)` is `(a - b)`, and not `(d - b)` since the conjecture is traversed in pre visit order.

$\underset{\sim\sim\sim}{\texttt{(-b + a)}}$ `+ (b - c) =` $\underset{\sim\sim\sim}{\texttt{(-b + a)}}$ `+ (d - b)`

The subterm `(a - b)` has another occurrence in the right hand side of the conjecture that is affected too. □

The second form of contextual pattern comes handy when the subterm of interest occurs immediately under a context that is easy to describe.

Example 6. Take the following conjecture:

```
0 = snd (0 * c, 0 * (a + b))
```

To prove this conjecture it is enough to use the annihilating property of `0` on `(a + b)` and compute away the `snd` projection. Unfortunately that property also applies to `(0 * c)`. We can easily identify `(0 * (a + b))` with the second form of contextual pattern, mentioning the context symbol `snd` and marking with `X` the argument we are interested in. The resulting command is thus `rewrite [X in snd (_, X)]mul0n`. □

A typical example of the last form is with the `set` command, that creates a local definition grabbing instances of the definendum in the conjecture.

Example 7. Take the following conjecture:

```
a + b = f (a^2 + b) - c
```

To make it more readable one may want to abbreviate with `n` the expression `(a^2 + b)`. The command `set n := (_ + b in X in _ = X)` binds to `n` all the occurrences of the first instance of the pattern `(_ + b)` in the right hand side only of the conjecture.

$$\texttt{a + b = }\underline{\texttt{f}}\ \underset{\sim}{\texttt{n}}\ \underline{\texttt{- c}}$$

Note that the pattern `(_ + b)` could also match `(a + b)` in the left hand side of the conjecture, but the `(in X in _ = X)` part of the contextual pattern focuses the right hand side only. From now on we will always underline with a straight line the subterm selected by the context part of a pattern (i.e., the subterm identified by the bound variable `X` in the previous example). □

In Section 2.3 we describe how the user can define shortcuts for commonly used contexts, and thus write the previous pattern as: `set n := (_ + b in RHS)`.

We give an example of the third ⟨*c-pattern*⟩ form together with the examples for ⟨*r-pattern*⟩s.

Rewrite Patterns ⟨*r-pattern*⟩. The `rewrite` command supports two more patterns obtained by prefixing the first two ⟨*c-pattern*⟩s with the `in` keyword. The intended meaning is that the pattern identifies all subterms of the specified context. Note that the `rewrite` command can always infer a redex from the shape of the rewrite rule. For example the `addrC` rule of Example 1 gives the redex pattern `(_ + _)`.

in ⟨***tpat***⟩ is the simplest form of rewrite pattern. The redex is inferred from the rewriting rule. The subterms affected are all the occurrences of the first instance of the redex inside all the occurrences of the first instance of ⟨*tpat*⟩.

in ⟨***ident***⟩ **in** ⟨***tpat***⟩ is quite similar to the last form of contextual pattern seen above, but the redex is not explicitly given but instead inferred from the rewriting rule. The subterms affected are all the occurrences of the first instance of the redex inside the subterms identified by ⟨*ident*⟩ in all the occurrences of the first instance of ⟨*tpat*⟩.

Example 8. The first form of ⟨*r-pattern*⟩ is handy when we want to focus on the subterms of a given context. Take for example the following conjecture:

```
f (a + b) (2 * (a + c)) + (c + d) + f a (c + d) = 0
```

The command `rewrite [in f _ _]addrC` focuses the matching of the redex inferred from the `addrC` lemma, `(_ + _)`, to the subterms of the first instance of the pattern `(f _ _)`. Thus the conjecture is changed into

```
f (b + a) (2 * (a + c)) + (c + d) + f a (c + d) = 0
```

If the user had in mind to exchange `a` with `c` instead, she could have used a pattern like `[in X in f _ X]addrC`, to focus the matching of the redex on the second argument of `f`, obtaining:

```
f (a + b) (2 * (c + a)) + (c + d) + f a (c + d) = 0
```

The last form of ⟨*c-pattern*⟩ could be used to focus on the last occurrence of `(c + d)`. The pattern `[_ + d as X in f _ X]` would first match the context substituting `(_ + d)` for `X`. The pattern `(f _ (_ + d))` focuses on the second occurrence of `f`, then the `X` identifier selects only its second argument that is exactly where the rewriting rule `addrC` is applied.

```
f (a + b) (2 * (a + c)) + (c + d) + f a (d + c) = 0
```

It is important to note that even if the rewrite proof command always infers a redex from the rewrite rule, a different redex can be specified using a ⟨*c-pattern*⟩. This is especially convenient when the inferred redex is masked by a definition, as in Example 3 .

2.2 Matching Order

In the previous examples we implicitly followed a precise order when matching the various ⟨*tpat*⟩s part of a ⟨*c-pattern*⟩ or ⟨*r-pattern*⟩. For example we always matched the context part first. We now make this order explicit.

⟨*tpat*⟩, ⟨*ident*⟩ in ⟨*tpat*⟩ All the subterms of the conjecture are matched against ⟨*tpat*⟩.

⟨*tpat*⟩$_1$ as ⟨*ident*⟩ in ⟨*tpat*⟩$_2$ All the subterms of the conjecture are matched against ⟨*tpat*⟩$_2$[⟨*tpat*⟩$_1$/⟨*ident*⟩].

⟨*tpat*⟩$_1$ in ⟨*ident*⟩ in ⟨*tpat*⟩$_2$ First, subterms of the conjecture are matched against ⟨*tpat*⟩$_2$. Then the subterms of the instantiation of ⟨*tpat*⟩$_2$ identified by ⟨*ident*⟩ are matched against ⟨*tpat*⟩$_1$.

in ⟨*ident*⟩ in ⟨*tpat*⟩ First, subterms of the conjecture are matched against ⟨*tpat*⟩. Then the subterms of the instantiation of ⟨*tpat*⟩ identified by ⟨*ident*⟩ are matched against the inferred redex (that is always present since this pattern has to be used with the `rewrite` proof command).

in ⟨*tpat*⟩ First, subterms of the conjecture are matched against ⟨*tpat*⟩. Then the instantiation of ⟨*tpat*⟩ is matched against the inferred redex.

If one of the first four patterns is used in conjunction with `rewrite`, the instance of the redex is then matched against the pattern inferred from the rewriting rule. The matching order is very relevant to predict the instantiation of patterns.

Example 9. For example in the pattern `((_ + _) in X in (_ * X))`, the matching of the sub pattern `(_ + _)` is restricted to the subterm identified by `X`. Take the following conjecture:

```
a + b + (a * ((a + b) * d)) = 0
```

The dash underlined subterm would be a valid instance of `(_ + _)` but is skipped since it does not occur in the right context. In fact `(_ * X)` is matched first. The subterm corresponding to `X` is `((a + b) * d)`. Then its subterms are matched against `(_ + _)` and the first, and only, occurrence is `(a + b)`. □

2.3 Recurring Contexts

Whilst being quite expressive, contextual patterns tend to be a bit verbose and quite repetitive. For example to focus on the right hand side of an equational conjecture, one may have to specify the pattern `(in X in _ = X)`.

With a careful use of the notational mechanism of Coq we let the user define abbreviations for common contexts, corresponding to the ⟨*ident*⟩ `in` ⟨*tpat*⟩ part of the pattern. The definition of the abbreviation `RHS` is as follows.

```
Notation RHS := (X in _ = X)%pattern.
```

There the notational scope `%pattern` interprets the infix `in` notation in a peculiar way, encoding in a non ambiguous way the context `(X in _ = X)` in a simple ⟨*tpat*⟩. Then, when the system parses `(in RHS)` as an instance of `in` ⟨*tpat*⟩ it recognizes the context encoded in ⟨*tpat*⟩ and outputs the abstract syntax tree for `in` ⟨*ident*⟩ `in` ⟨*tpat*⟩.

3 Term Matching

We now give a precise description of the matching operation for ⟨*tpat*⟩. The main concerns are performances and predictability.

Predictability has already been discussed in relation to Example 3. A lemma that talks about the `map` function should affect occurrences of the `map` function only, even if other subterms are *defined* in terms of `map`, unless the user really means that. Indeed the most characterizing feature of the logic of Coq is to identify terms up to definition unfolding and computation. That allows to completely omit proof steps that are pure computations, for example `(0 + x)` and `x` are just equal (not only provably equal) for the standard definition of addition.

Performance is a main concern when one deals with large conjectures. To take advantage of the computational notion of comparison the logic offers, one could be tempted to try to match the pattern against any subterm, even if the subterm shares no similarity with the pattern itself. A higher order matching procedure could find that the pattern actually matches up to computation. Nevertheless, this matching operation could be expensive. Especially because it is expected to fail on most of the subterms and failure is certain only after both the pattern and the subterm are reduced to normal forms.

The considerations about performances and predictability lead to the idea of *keyed matching*. The matching operation is attempted only on subterms whose head constant is equal to the head constant (the *key*) of the pattern, *verbatim*. Arguments of the key are matched using the standard higher order matching algorithm of Coq, which takes computation into account.

Take for example the conjecture (*huge* `+ x * (1 - 1) = 0`) and the rewrite rule `muln0` that gives the redex `(_ * 0)`. The key of the redex `*` is compared with the head of all the subterms of the conjecture. This traversal is linear in size of the conjecture. The higher order matching algorithm of Coq is thus run on the candidate subterms identified by the keyed filtering phase, like `(x * (1 - 1))`. In that case the second argument of the pattern, `0`, matches `(1 - 1)` up to reduction. The *huge* subterm, assuming it contains no `*`, is quickly skipped, and the expensive but computation aware matching algorithm of Coq is never triggered on its subterms.

3.1 Gadgets

To adhere to the keyed matching discipline, that is different from the standard one implemented in Coq, we had to implement our own stricter matching algorithm inside the SSReflect extension, piggybacking on Coq's general unification procedure. This gave us the opportunity to tune it towards our needs, adding some exceptions for unit types, abstract algebraic structures, etc.

Unit types are types that have only one canonical inhabitant. In a library of the extent of the SSReflect's one, there are many of them. For example there is only one matrix of 0 rows or 0 columns, there is only one natural number in the interval subtype `[0,1[`, there is only one 0-tuple, etc.

In the statement of the canonicity lemma for these types, the inferred redex is just a wild card, i.e., there is no key. In the following example the type `'I_n` denotes the subtype of natural numbers strictly smaller than `n`.

```
Lemma ord1 (x : 'I_1) : x = 0.
```

A pattern with no key results in a error message in SSReflect. Nevertheless SSReflect supports a special construction to mark wild cards meant to act as a key. In that case the search is driven by the type of the wild card. In the following example the `(unkeyed x)` notation denotes any subterm of type `'I_1`.

```
Notation unkeyed x := (let flex := x in flex).
Lemma ord1 (x : 'I_1) : unkeyed x = 0.
```

Another notable exception is the case in which the key is a projection. The logic of Coq can represent dependently typed records [15], that are non homogeneous n-tuples where the type of the i-th element can depend on the values of the previous $i - 1$ elements. This is a key device to model abstract algebraic structures [16,8,17,6], like a `Monoid` as a record with three fields: a type `mT`, a binary operation `mop` on `mT` and the associative property for `mop`[1].

```
Structure Monoid := {         mT (M : Monoid) : Type
  mT : Type;                  mop (M : Monoid) : mt M -> mt M -> mt M
  mop : mT -> mT -> mT;       massoc (M : Monoid) (x y z : mt M) :
  massoc : assoc mop }          mop M x (mop M y z) = mop M (mop M x y) z
```

[1] We omit the unit for reasons of space.

Constants `mT`, `mop` and `massoc` are projections for the corresponding record fields. Their types are reported on the right.

If we look at the statement of any lemma equating `Monoid` expressions we note that the key for the operation is `mop`, as in the statement of `massoc` that leads to the pattern `(mop _ _ (mop _ _ _))`.
Algebraic reasoning is interesting because all the results proved in the abstract setting apply to any instance of the algebraic structure. For example lists and concatenation form a `Monoid`. Nevertheless, any conjecture about lists is going to use the concrete concatenation operation `cat`. The higher order matching algorithm of Coq can be instrumented to exploit the fact that there exists a canonical `Monoid` over lists and is thus able to match `(mop _ _ _)` against `(cat s1 s2)` assigning to the first wild card this canonical `Monoid` structure. Unfortunately, our matching algorithm would fail to match any occurrence of `cat` against the key `mop`, because they not equal verbatim.
The exception to the keyed matching discipline we considered is to compare as verbatim equal keys that happen to be projections with any of their canonical values. For example the key `mop` will match list concatenation, but also integer addition etc. and any other operation that is declared to form a `Monoid`. Note that this matching requires to correctly align the pattern with the term to be matched. In case of the term `(cat s1 s2)`, the pattern `(mop _ _ _)` has to be matched as follows: the `cat` term has to be matched against the initial part of the pattern `(mop _)`, that corresponds to the projection applied to the unknown `Monoid` structure. Then the following two arguments `s1` and `s2` have to be matched against the two wild cards left.

The last exception is for patterns with a flexible key but some arguments, like `(_ a b)`. The intended meaning is that the focus is on the application of a function whose last two arguments are `a` and `b`. This kind of pattern lacks a key and its match is attempted on any subterms. This is very convenient when the head constant of the expression to be focused is harder to write than the arguments. For example the expression `([predI predU A B & C] x)` represents the application of a composite predicate to `x`. This expression can be easily identified with the pattern `(_ x)`.

3.2 Verbatim Matching of the Pattern

There is an important exception to the keyed matching discipline worth explaining in more details. We begin with a motivating example, showing a situation in which the keyed matching prevents the user from freely normalizing associativity.

Example 10 (Motivating example)

```
Lemma example n m : n + 2 * m = m + (m + n)
by rewrite addnA addnC !mulSn addn0.
```

Without the verbatim matching phase, the application of the first rewrite rule, `addnA`, would turn the conjecture into:

```
n + m + (1 * m) = m + (m + n)
```

In fact, the redex inferred from `addnA` is `(_ + (_ + _))`, and the first occurrence of its key `+` is in the left hand side of the conjecture. Since the definition of multiplication for natural numbers is computable Coq compares `(2 * m)` and `(m + (1 * m))` as equal. Thus `(n + (m + (1 * m)))` is successfully matched against the redex failing the user expectations. Thus the user is forced to add a pattern like `(m + _)` to `addnA` that is somewhat unnatural, since there is only one visible occurrence of the redex `(_ + (_ + _))`. □

To address this issue, the term pattern matching algorithm performs two phases. In the first no reduction takes place, and syntactic sugar, like hidden type arguments or explicit type casts, is taken into account, essentially erasing invisible arguments from the pattern and the conjecture. The second phase is driven by the pattern key and allows full reduction on its arguments.

Once the pattern is instantiated the search for its occurrences is again keyed, and arguments are compared pairwise allowing conversion. For example consider the pattern `(_ + _)` and its first instance `(1 + m)`. Other occurrences are searched using `+` as the key, and arguments are compared pairwise. Thus a term like `(0 + (1 + m))`, that is indeed computationally equal to `(1 + m)`, is not an occurrence of `(1 + m)`, since `1` does not match `0` and `m` does not match `(1 + m)`. On the contrary `(1 + (0 + m))` is an occurrence of `(1 + m)`.

4 Related Works

In order to compare the approach proposed in this paper with other approaches we consider the following conjecture `f (a + 0) = a + 0 * b` where we want to replace `(a + 0 * b)` with `a` using the equation `forall x, x + 0 = x` named `addn0`. Note that the logic of Coq compares `(0 * b)` and `0` as equal.

The first comparison that has to be made is with the standard Coq mechanism to focus on subexpressions. The `pattern` command ([5, Section 6.3.3]) pre-processes the conjecture putting in evidence the sub term of interest. The user has to spell out completely this subexpression, and if it occurs multiple times she can specify occurrence numbers. This leads to the proof script: `pattern (a + 0 * b); rewrite addn0`. Note that the two expressions `(a + 0)` and `(a + 0 * b)` are not consider as equal by the matching algorithm used by `pattern` and by Coq's `rewrite`. For example `pattern (a + 0) at 2` fails as well as `rewrite (addn0 a) at 2`. We believe our approach is superior because the intent of the user is not obfuscated by commands to prepare the conjecture and because it takes computation into account when comparing terms. In SSReflect one can perform the desired manipulation with `rewrite [RHS]addn0` where `RHS` is a pattern notation as in Section 2.3.

Matita [3] is an ITP based on the same logic of Coq with a modern graphical interface. Its proof language is similar to the one of Coq but (parts) of proof commands can be generated by mouse gestures. In particular the user can focus on a sub term selecting it with the mouse. The selection of the right hand side of the running example results in the following snippet `rewrite addOn in |- ???%` where `%` marks the subterm of interest, while the three `?` respectively stand for the head symbol (`=`), the invisible type parameter `nat` and the left hand side, all of which have to be ignored by the `rewrite` tactic. While this approach is intuitive and effective in writing proof scripts, it does not ease their maintenance, since the textual representation of the visual selection is not very informative for the reader, especially when selection happens deep inside the conjecture.

The ISABELLE prover [14] implements a framework on top of which different logics and proof languages are built. The most used combination is higher order logic and the declarative proof language Isar [19]. In this setting some of the complexity introduced by the logic of COQ disappears. For example terms are not considered to be equal taking definitions into account. Moreover in the declarative style proposed by ISAR language the user spells out the conjecture more frequently, and some automation tries to prove it, finding for example which occurrences need to be rewritten. To avoid repeating large expressions the Isar language provides the `(is` ⟨*pattern*⟩`)` construct [19, Section 3.2.6] to bind expressions to schematic variables that can be reused later on.

5 Conclusion

This paper presents the language of patterns adopted by the SSREFLECT proof shell extension for the COQ system. The language was introduced in SSREFLECT version 1.3 in March 2011 mainly to improve the effectiveness of the `rewrite` proof command. Version 1.4 makes the pattern language consistently available to all language constructs that have to identify subexpressions of the conjecture.

The choices made in the design of the pattern language and its semantics are based on the experience gathered in the Mathematical Components team on the formal proof of the Odd Order Theorem during the last five years, and the implementation has been validated by roughly one year of intense use. As of today this formalization comprises 113,384 lines of code, of which 34,077 contain a `rewrite` statement. Of these 2,280 have been changed to take advantage of the pattern language, and some other 2,005 lines (of which 1,705 contain a `rewrite` command) still make use of occurrence numbers and could be modified too.

A line of ongoing development is to separate the code implementing the pattern matching algorithm and the parsing of patterns concrete syntax from the rest of the SSREFLECT extension, making a separate extension. This will allow proof commands provided by other COQ extensions to benefit from the same pattern language. A possible application is in the AAC COQ extension that automates proofs dealing with associativity and commutativity. In fact in [7] Braibant and Pous express the need for a linguistic construct to select subexpressions other than occurrence numbers.

We thank Frédéric Chyzak for some very productive discussions on the topic.

References

1. Mathematical Components website,
`http://www.msr-inria.inria.fr/Projects/math-components/`,
`http://coqfinitgroup.gforge.inria.fr/ssreflect-1.3/`
2. 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)
3. Asperti, A., Coen, C.S., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Journal of Automated Reasoning 39(2), 109–139 (2007)
4. Bender, H., Glauberman, G.: Local analysis for the Odd Order Theorem. London Mathematical Society Lecture Note Series, vol. 188. Cambridge University Press (1994)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions. Springer (2004)
6. 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)
7. Braibant, T., Pous, D.: Tactics for Reasoning Modulo AC in COQ. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 167–182. Springer, Heidelberg (2011)
8. 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)
9. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection extension for the COQ system. INRIA Technical report, 00258384
10. Gonthier, G.: Formal proof – the four color theorem. Notices of the American Mathematical Society 55, 1382–1394 (2008)
11. 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)
12. Huppert, B., Blackburn, N.: Finite groups II. Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete, vol. 2. Springer (1982)
13. The COQ development team. The COQ proof assistant reference manual, Version 8.3 (2011)
14. Paulson, L.C.: The foundation of a generic theorem prover. Journal of Automated Reasoning 5, 363–397 (1989)
15. Pollack, R.: Dependently typed records in type theory. Formal Aspects of Computing 13, 386–402 (2002)
16. Sacerdoti Coen, C., Tassi, E.: Working with Mathematical Structures in Type Theory. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 157–172. Springer, Heidelberg (2008)
17. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science 21, 1–31 (2011)
18. Théry, L., Hanrot, G.: Primality Proving with Elliptic Curves. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 319–333. Springer, Heidelberg (2007)
19. Wenzel, M.T.: Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–184. Springer, Heidelberg (1999)

Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL

Fabian Immler and Johannes Hölzl[*]

Institut für Informatik, Technische Universität München
www.in.tum.de/~immler, www.in.tum.de/~hoelzl

Abstract. Many ordinary differential equations (ODEs) do not have a closed solution, therefore approximating them is an important problem in numerical analysis. This work formalizes a method to approximate solutions of ODEs in Isabelle/HOL.

We formalize initial value problems (IVPs) of ODEs and prove the existence of a unique solution, i.e. the Picard-Lindelöf theorem. We introduce generic one-step methods for numerical approximation of the solution and provide an analysis regarding the local and global error of one-step methods.

We give an executable specification of the Euler method as an instance of one-step methods. With user-supplied proofs for bounds of the differential equation we can prove an explicit bound for the global error. We use arbitrary-precision floating-point numbers and also handle rounding errors when we truncate the numbers for efficiency reasons.

Keywords: Formalization of Mathematics, Ordinary differential equation, Numerical Analysis, One-Step method, Euler method, Isabelle/HOL.

1 Introduction

Ordinary differential equations (ODEs) have a lot of important applications. They are for example used to describe motion or oscillation in Newtonian mechanics, the evolution or growth of organisms in biology, or the speed of chemical reactions.

The Picard-Lindelöf theorem states the existence of a unique solution (under certain conditions) but unfortunately, many problems do not allow an explicit closed formula as solution (e.g. the seemingly simple ODE $\dot{x} = x^2 - t$ for initial values $x(t_0) = x_0$). In such cases, one has to content oneself with numerical methods that give approximations to the solution.

In order to evaluate the quality of an approximate solution (which depends very much on the concrete problem) you need to choose the parameters of your numerical method (i.e. step size, precision) wisely. This is where the use of an interactive theorem prover might be useful: We formalize initial value problems (IVPs) of ODEs and prove the existence of a unique solution in Isabelle/HOL. We give an executable specification of the Euler method – a basic numerical algorithm – and prove the error bound of the approximation.

[*] Supported by the DFG Graduiertenkolleg 1480 (PUMA).

2 Related Work

When an ODE has a solution representable in a closed form, an approximation method for this closed form can be used. Muñoz and Lester [12] use rational interval arithmetic in PVS to efficiently approximate real valued functions in theorem provers. In addition to basic arithmetic operations they also support trigonometric functions. Melquiond [11] implements a similar method in Coq. He also implements interval arithmetic, but uses floating-point numbers and sophisticated methods to avoid loss of correlation.

An alternative to interval arithmetic is to approximate real numbers by a sequence of rational numbers. Each element in this sequence has a defined distance to the exact result. Harrison [5] uses this approach to compute the logarithm. O'Connor [15] approximates real numbers by organizing their completion of rational numbers in a monad. O'Connor and Spitters [14] use this monadic construction in order to implement arbitrary approximations of the Riemann integral. Krebbers and Spitters [9,10] extend this work to use arbitrary-precision floating-point numbers. Similar to the proof of the existence of the unique solution of an IVP, one can iterate an integral operation in order to approximate the solution (as suggested in [17]).

Boldo *et al.* [1] formalize partial differential equations stemming from acoustic wave equations in Coq. As they analyse partial differential equations they can not show a general existence or uniqueness theorem. Their particular problem admits an analytical solution and they simply assume that the solution is unique. They also show consistency and stability and that in their case convergence follows. However, they needed to find an analytical formula for the rounding errors.

3 Preliminaries

3.1 Isabelle/HOL

The formalizations presented in this paper are done in the Isabelle/HOL theorem prover. In this section we give an overview of our syntactic conventions.

The term syntax follows the λ-calculus, i.e. function application is juxtaposition as in $f\ t$ and function abstraction is written as $\lambda x.\ t$. The notation $t :: \tau$ means that t has type τ. Types are built from base types like $\mathbb{N}$ (natural numbers), $\mathbb{R}$ (real numbers), $\mathbb{R}^n$ (Euclidean spaces of dimension n), type variables (α, β, etc.), functions $\alpha \to \beta$, sets $\mathcal{P}(\alpha)$, and pairs $\alpha \times \beta$.

Our work builds on the `Multivariate_Analysis` library which was ported from Harrison's Euclidean spaces for HOL-Light [6]. In our formalization the Euclidean space $\mathbb{R}^n$ is not just the function space $n \to \mathbb{R}$; it is a type class denoting a Euclidean space. $\mathbb{R}^n \times \mathbb{R}^m$, $\mathbb{R}$, and $n \to \mathbb{R}$ are in this type class.

We write $(a, b) :: \alpha \times \beta$ for pairs, $A \times B := \{(a, b) \mid a \in A \wedge b \in B\} :: \mathcal{P}(\alpha \times \beta)$ for the Cartesian product of A and B (do not confuse with the product type), $\|x\|$ for the norm of x, $\mathcal{B}_r(x) := \{y \mid \|x - y\| \le r\} :: \mathcal{P}(\mathbb{R}^n)$ for the closed ball around x with radius $r :: \mathbb{R}$, and *sup* A and *inf* A for the supremum and infimum of A. With $\dot{x}(t) = y$ we denote that $x :: \mathbb{R} \to \mathbb{R}^n$ has at t the derivative $y :: \mathbb{R}^n$, and with $\int_a^b f\ x\ dx$ we denote the integral of $f :: \mathbb{R}^n \to \mathbb{R}^m$ over $[a;b]$. Hilbert choice is $(\varepsilon\, x.\ P\ x)$, i.e. $(\exists x.\ P\ x) \Rightarrow P(\varepsilon\, x.\ P\ x)$ holds. In this section x_i is the i-th projection of the vector x. We write $[a;b] := \{x \mid \forall i.\ a_i \leq x_i \wedge x_i \leq b_i\} :: \mathcal{P}(\mathbb{R}^n)$ for hyperrectangles on Euclidean spaces (which are closed intervals on $\mathbb{R}$), and $\mathcal{R}_r(x) := \{y \mid \forall i.\ y_i \in [x_i - r; x_i + r]\}$ for hypercubes around x. The predicate $\textit{is-interval}\ S := (\forall a, b \in S.\ \forall x.\ (\forall i.\ x_i \in [a_i; b_i] \cup [b_i; a_i]) \Rightarrow x \in S)$ accepts intervals in general, mixtures of open and closed, and infinite ones.

3.2 Arbitrary-Precision Floating-Point Numbers ($\mathbb{F}$)

The previous formalization of arbitrary-precision floating-point numbers in Isabelle/HOL [13,7] used pairs of integer exponents e and mantissas m, representing the real numbers $m \cdot 2^e$. Unfortunately this results in a type which has multiple representations for the same number, e.g. zero is represented by $0 \cdot 2^e$ for every e. Therefore the resulting type does not support any interesting type class, like linear order, commutative groups for addition, etc.

Hence, we introduce a new type $\mathbb{F}$ as the dyadic rationals, i.e. all numbers x which are representable as $m \cdot 2^e$. We have an injective function $(\cdot)_\mathbb{R} :: \mathbb{F} \to \mathbb{R}$ and its partially specified inverse $(\cdot)_\mathbb{F} :: \mathbb{R} \to \mathbb{F}$. As (non-injective) constructor we introduce $\textit{Float}\ m\ e = (m \cdot 2^e)_\mathbb{F}$ and declared it as a datatype constructor for code generation [3]. We lift the arithmetic constants 0, 1, $+$, $-$, $\cdot$, $<$, $\leq$ from the reals and provide executable equations, e.g. for multiplication:

$$(\textit{Float}\ m_1\ e_1) \cdot (\textit{Float}\ m_2\ e_2) = \textit{Float}\ (m_1 \cdot m_2)\ (e_1 + e_2).$$

3.3 Bounded Continuous Functions

The proof for the existence of a unique solution to an IVP is based on an application of the Banach fixed point theorem, which guarantees the existence of a unique fixed point of a contraction mapping on metric spaces. The textbook-proof of Walter [18] defines a metric space on continuous functions with a compact domain. As functions in Isabelle/HOL are required to be total, one cannot simply restrict the domain, hence a slightly different approach is necessary: We define a type $\overline{\mathcal{C}}$ carrying bounded continuous functions, i.e. functions which are continuous everywhere and whose values are bounded by a constant:

$$\overline{\mathcal{C}} = \{f :: \mathbb{R}^n \to \mathbb{R}^m \mid f \text{ continuous on } \mathbb{R}^n \wedge (\exists B.\ \forall t.\ \|f\ t\| \leq B)\}$$

The morphisms $\textit{Rep}_{\overline{\mathcal{C}}} : \overline{\mathcal{C}} \to (\mathbb{R}^n \to \mathbb{R}^m)$ and $\textit{Abs}_{\overline{\mathcal{C}}} : (\mathbb{R}^n \to \mathbb{R}^m) \to \overline{\mathcal{C}}$ allow to use an element of type $\overline{\mathcal{C}}$ as function and to define elements of type $\overline{\mathcal{C}}$ in terms of a function. We define a norm on $\overline{\mathcal{C}}$ as the supremum of the range and operations $+$, $-$, $\cdot$ pointwise.

$$\begin{aligned}
\|f\| &:= \textit{sup}\ \{\|\textit{Rep}_{\overline{\mathcal{C}}}\ f\ x\| \mid x \in \mathbb{R}^n\} \\
f + g &:= \textit{Abs}_{\overline{\mathcal{C}}}(\lambda x.\ f\ x + g\ x) \\
f - g &:= \textit{Abs}_{\overline{\mathcal{C}}}(\lambda x.\ f\ x - g\ x) \\
a \cdot f &:= \textit{Abs}_{\overline{\mathcal{C}}}(\lambda x.\ a \cdot f\ x)
\end{aligned}$$

We prove that $\overline{\mathcal{C}}$ is a normed vector space, hence also a metric space. In order to be able to apply the Banach fixed point theorem we need to show that $\overline{\mathcal{C}}$ is a complete space, meaning that every Cauchy sequence converges. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ that is continuous on a compact interval $[a; b]$ is converted to $\overline{\mathcal{C}}$ by extending the function continuously outside the domain with the help of *clamp*:

$$\begin{aligned}&(\mathit{clamp}_{[a;b]}\ x)_i := \textbf{\textit{if}}\ x_i \le a_i\ \textbf{\textit{then}}\ a_i\ \textbf{\textit{else}}\ (\textbf{\textit{if}}\ x_i \ge b_i\ \textbf{\textit{then}}\ b_i\ \textbf{\textit{else}}\ x_i)\\ &\mathit{ext\text{-}cont}_{[a;b]}\ f := \mathit{Abs}_{\overline{\mathcal{C}}}(\lambda x.\ f\ (\mathit{clamp}_{[a;b]}\ x))\end{aligned}$$

The key property we use is that an extended function is continuous everywhere when it was continuous on the interval. Inside the interval the resulting function takes the same values as the original function:

$$f \text{ continuous on } [a; b] \Rightarrow \mathit{Rep}_{\overline{\mathcal{C}}}(\mathit{ext\text{-}cont}_{[a;b]} f) \text{ continuous on } \mathbb{R}^n$$
$$x \in [a; b] \Rightarrow \mathit{Rep}_{\overline{\mathcal{C}}}(\mathit{ext\text{-}cont}_{[a;b]} f)\ x = f\ x$$

4 Initial Value Problems

4.1 Definition

An equation in which an unknown function $u :: \mathbb{R} \to \mathbb{R}^n$ and derivatives of this function occur is an ODE. ODEs with derivatives of higher order can be reduced to a first order system, which is why we handle ODEs of first order only. An ODE together with values t_0, x_0 for an initial condition $u\ t_0 = x_0$ is an IVP and can always be written in terms of a right-hand side f which is supposed to be defined on a domain $I \times D$ (Compare with Figure 1):

$$\dot{u}\ t = f(t, u\ t) \qquad u\ t_0 = x_0 \in D \qquad t_0 \in I$$

We define IVPs in Isabelle as a record, which is a named tuple of elements

$$f :: \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n, \quad t_0 :: \mathbb{R}, \quad x_0 :: \mathbb{R}^n, \quad I :: \mathcal{P}(\mathbb{R}), \quad D :: \mathcal{P}(\mathbb{R}^n)$$
$$\mathit{ivp} = (f, t_0, x_0, I, D).$$

For the rest of the paper, we assume an IVP *ivp* with $(t_0, x_0) \in I \times D$ and use the symbols f, t_0, x_0, I, and D without further notice as part of this IVP.[1] We will notate modified IVPs in subscripts, for example $\mathit{ivp}_{f:=g}$ for the IVP *ivp* with the right-hand side g instead of f. Other components are updated analogously. Later definitions implicitly depend on *ivp* and may be updated in a similar fashion.

4.2 Solutions

We capture the notion of a solution to an IVP in the predicate *is-solution*: A solution needs to satisfy the initial condition and the derivative has to be given by f. Apart from that, we need to state explicitly that the potential solution must not leave the codomain. Mathematicians usually do that implicitly when declaring f as a function with domain $I \times D$.

$$\mathit{is\text{-}solution}\ u := u\ t_0 = x_0 \wedge (\forall t \in I.\ \dot{u}\ t = f(t, u\ t) \wedge u\ t \in D)$$

[1] In Isabelle/HOL: *ivp* is fixed as a locale parameter.

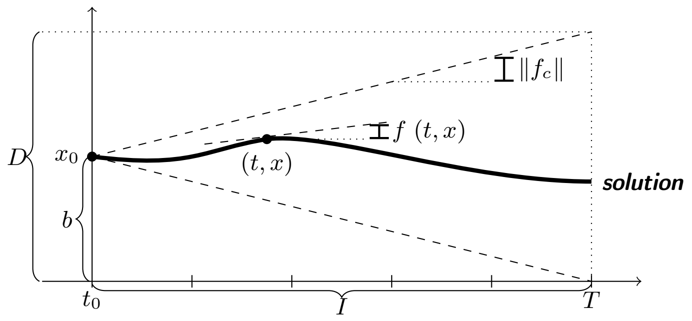


Fig. 1. The solution of an IVP on a rectangular domain and related variables

unique-solution. The Picard-Lindelöf theorem states the existence of a unique solution. We formalize the notion of a unique solution as follows: If two functions are solutions, they must attain the same values on I. We use Hilbert choice to obtain *solution* and relate all upcoming facts about solutions to IVPs to it.

$$\begin{aligned} \textit{has-solution} &:= (\exists u.\ \textit{is-solution}\ u) \\ \textit{solution} &:= (\varepsilon u.\ \textit{is-solution}\ u) \\ \textit{unique-solution} &:= \textit{has-solution} \wedge (\forall v.\ \textit{is-solution}\ v \Rightarrow (\forall t \in I.\ v\ t = \textit{solution}\ t)) \end{aligned}$$

4.3 Combining Initial Value Problems

Working with IVPs in a structured way helped us to implement the proofs in a maintainable fashion. An important operation is to be able to "connect" two solutions at a common point. We therefore assume two IVPs $\textit{ivp}_1$ and $\textit{ivp}_2$ that we want to combine to an IVP $\textit{ivp}_c$:

$$\begin{aligned} \textit{ivp}_1 &= (f_1, t_0, x_0, [t_0; t_1], D) \\ \textit{ivp}_2 &= (f_2, t_1, \textit{solution}_{\textit{ivp}_1}\ t_1, [t_1; t_2], D) \\ f_c &:= (\lambda(t,x).\ \textbf{\textit{if}}\ t \le t_1\ \textbf{\textit{then}}\ f_1(t,x)\ \textbf{\textit{else}}\ f_2(t,x)) \\ \textit{ivp}_c &:= (f_c, t_0, x_0, [t_0; t_2], D) \end{aligned}$$

Assuming unique solutions for $\textit{ivp}_1$ and $\textit{ivp}_2$, we prove a unique solution for $\textit{ivp}_c$:

$$\begin{aligned} & f_1(t_1, \textit{solution}_{\textit{ivp}_1}\ t_1) = f_2(t_1, \textit{solution}_{\textit{ivp}_1}\ t_1) \Rightarrow \\ & \quad \textit{unique-solution}_{\textit{ivp}_1} \Rightarrow \textit{unique-solution}_{\textit{ivp}_2} \Rightarrow \textit{unique-solution}_{\textit{ivp}_c} \end{aligned}$$

4.4 Quantitative Picard-Lindelöf

In this section, we show that certain sets of assumptions (*bnd-strip*, *strip*, *rect*) imply *unique-solution*, i.e. the existence of a unique solution and therefore several variants of the Picard-Lindelöf theorem[2]. We will present key parts of the proofs

[2] In Isabelle/HOL: We show that e.g. *rect* is a sublocale of *unique-solution*.

and how they have been implemented in Isabelle, especially to show how our choice of formalization helped to structure the proofs. The proofs in this section are inspired by Walter [18] and follow closely the structure which is given there. All of the proofs provide concrete results that we use later in the numerical approximation of IVPs.

Lipschitz continuity is a basic assumption for the Picard-Lindelöf theorem. It is stronger than continuity: it limits how fast a function can change. A function g is (globally) Lipschitz continuous on a set S if the slope of the line connecting any two points on the graph of g is bounded by a constant:

$$\textsf{\textit{lipschitz}}\ S\ g\ L := (\forall x, y \in S.\ \|g\ x - g\ y\| \leq L \cdot \|x - y\|)$$

bnd-strip. If we choose $I = [t_0; T]$ and $D = \mathbb{R}^n$ for the domain of a function f that is continuous on $I \times D$ and assume a global Lipschitz constant L on D, we can show the existence of a unique solution, provided that $(T - t_0) \cdot L < 1$ holds. We call this set of assumptions *bnd-strip*:

$$\begin{aligned}\textsf{\textit{bnd-strip}}\ T\ L := {} & f \text{ continuous on } I \times D\ \wedge \\ & (\forall t \in I.\ \textsf{\textit{lipschitz}}\ D\ (\lambda x.\ f(t, x))\ L)\ \wedge \\ & I = [t_0; T] \wedge D = \mathbb{R}^n \wedge (T - t_0) \cdot L < 1\end{aligned}$$

Using the fundamental theorem of calculus, any solution to the equalities of the IVP must also satisfy $u\ t - x_0 = \int_{t_0}^{t} f(\tau, u\ \tau) \mathsf{d}\tau$ for all t in I. This equality can be seen as an iteration of functions, known as *Picard iteration* which we conduct in the space of bounded continuous functions (moving explicitly between functions $\mathbb{R} \to \mathbb{R}^n$ and $\overline{\mathcal{C}}$ is the most prominent difference to the textbook proof):

$$\mathsf{P}\ x := \textsf{\textit{ext-cont}}_{[t_0;T]} \left(\lambda t.\ x_0 + \int_{t_0}^{t} f\big(\tau, \textsf{\textit{Rep}}_{\overline{\mathcal{C}}}\ x\ \tau\big) \mathsf{d}\tau \right); \qquad \mathsf{P} :: \overline{\mathcal{C}} \to \overline{\mathcal{C}}$$

In this metric space, we show that P is Lipschitz continuous for the constant $(T - t_0) \cdot L$. The Lipschitz constant is less than 1 (i.e. $(T - t_0) \cdot L < 1$). This is a necessary assumption – together with the completeness of $\overline{\mathcal{C}}$ – to apply the Banach fixed point theorem (from the `Multivariate_Analysis` library). It guarantees the existence of a unique fixed point x^* for the mapping P.

Together with the fundamental theorem of calculus we show that the fixed point x^* of P is a solution. Moreover every (continuously extended) solution u is a fixed point of P

$$\textsf{\textit{is-solution}}\ (\textsf{\textit{Rep}}_{\overline{\mathcal{C}}}\ x^*)$$

$$\textsf{\textit{is-solution}}\ u \Rightarrow \mathsf{P}(\textsf{\textit{ext-cont}}_{[t_0;T]}\ u) = \textsf{\textit{ext-cont}}_{[t_0;T]}\ u$$

from which we conclude the existence of a unique solution:

Theorem 1 (Picard-Lindelöf). *If f is continuous and Lipschitz continuous in its second variable and the interval $I = [t_0; T]$ is small enough, then there exists a unique solution:*

$$\textsf{\textit{bnd-strip}}\ T\ L \Rightarrow \textsf{\textit{unique-solution}}.$$

strip. According to *bnd-strip*, the size of the interval $[t_0; T]$ in which we have proven the existence of a unique solution depends on the Lipschitz-constant L.

In *strip* we drop the restriction $T - t_0 < \frac{1}{L}$ to the size of the interval. This can be done by splitting the desired interval $[t_0; T]$ into n sub-intervals, such that $\frac{T-t_0}{n}$ is small enough to satisfy the assumptions of *bnd-strip*. In an inductive (on $n \geq 1$) proof, one has (as hypothesis) the existence of a *unique-solution* on $[t_0; T - \frac{1}{n}(T - t_0)]$. The interval $[T - \frac{1}{n}(T - t_0); T]$ satisfies the assumptions for *bnd-strip* – consequently we have a *unique-solution* there, too. The respective solutions can then be combined. The argumentation in the textbook proof relies on geometric intuition when one combines solutions – doing this formally requires more efforts, but section 4.3 helped in retaining structure in the proofs.

rect. In *strip*, there is the assumption $D = \mathbb{R}^n$ for the codomain of the solution. One might want to restrict this part – e.g. if there is no Lipschitz constant on the whole codomain – to $D = \mathcal{R}_b(x_0)$. In this case (which we will call *rect*) we continuously extend the right-hand side f outside the rectangle to f_c:

$$f_c := \textit{ext-cont}_{[t_0;T] \times \mathcal{R}_b(x_0)}\ f; \qquad\qquad f_c :: \overline{\mathcal{C}}$$

The textbook also works with a continuous extended function, but we do so more explicitly with the utilization of *ext-cont*. We used *ext-cont* to obtain f_c, hence $\textit{Rep}_{\overline{\mathcal{C}}}\ f_c$ is continuous on the whole domain $I \times \mathbb{R}^n$. We apply Theorem 1 to obtain the existence of a unique solution for $\textit{Rep}_{\overline{\mathcal{C}}}\ f_c$. We show that the solution does not leave the codomain D – to ensure that $f = \textit{Rep}_{\overline{\mathcal{C}}}\ f_c$. For this, one has to choose a small enough upper bound T of the existence interval $[t_0; T]$. This depends on the maximum slope of the solution which is bounded by $\|f_c\|$, see Fig. 1. The formal proof centers around an application of the mean value theorem, this is tedious compared to the geometric intuition given in the textbook.

$$\textit{rect}\ T\ b\ L := f \text{ continuous on } I \times D \wedge (\forall t \in I.\ \textit{lipschitz}\ D\ (\lambda x.\ f(t,x))\ L) \wedge$$
$$I = [t_0; T] \wedge D = \mathcal{R}_b(x_0) \wedge b \geq 0 \wedge T \leq t_0 + b/\|f_c\|$$

Under these assumptions, we show that any solution to *ivp* cannot leave D.

$$\textit{rect}\ T\ b\ L \Rightarrow \textit{is-solution}\ u \Rightarrow (\forall t \in I.\ u\ t \in D)$$

Having this, we can show that $\textit{solution}_{f:=f_c}$ is a solution to *ivp* and that every other solution to *ivp* is also a solution to $\textit{ivp}_{f:=f_c}$. Consequently:

Theorem 2 (Picard-Lindelöf on a restricted domain)

$$\textit{rect}\ T\ b\ L \Rightarrow \textit{unique-solution}$$

4.5 Qualitative Picard-Lindelöf

In this section, we present a variant of the Picard-Lindelöf theorem (following the textbook proof of Walter [18] closely), which is mainly of mathematical interest: One does not get explicit values that could be used to estimate the error in a numerical approximation – which is what we need in the upcoming sections.

local-lipschitz. Many functions do not have a global Lipschitz constant L (e.g. $f(t,x) = x^2$ on $\mathbb{R}$). The weaker assumption of *local Lipschitz continuity* allows to prove the existence of a solution in a neighborhood of the initial value. A function f is locally Lipschitz continuous in its second variable if for every point (t,x) of the domain, there exists a neighborhood $\mathcal{B}_\epsilon(t,x)$ inside which there exists a Lipschitz constant L:

$$\begin{aligned}\textsf{\textit{local-lipschitz}} := \forall(t,x) \in I \times D.\ \exists\epsilon > 0.\ \exists L.\\ \forall u \in \mathcal{B}_\epsilon(t) \cap I.\ \textsf{\textit{lipschitz}}\ (\lambda x. f(u,x))\ (\mathcal{B}_\epsilon(x) \cap D)\ L\end{aligned}$$

open-domain. Together with the notion of local Lipschitz continuity, we get a very general result for the existence of a unique solution if we assume an open domain. We will use the set of assumptions *open-domain* to prove the existence of a unique solution on a maximal existence interval.

$$\textsf{\textit{open-domain}} := \textsf{\textit{local-lipschitz}} \wedge \textsf{\textit{open}}\ I \wedge \textsf{\textit{open}}\ D$$

Under these assumptions, we construct a small enough rectangle inside a neighborhood of the initial values that is inside the domain and possesses a Lipschitz-constant. From this we can conclude

$$\exists T > t_0.\ [t_0;T] \subseteq I \wedge \textsf{\textit{unique-solution}}_{I:=[t_0;T]}.$$

We define $\varPhi$ (similar to the textbook, but more explicit) to be the set of all solutions to *ivp* and upper bounds of their existence intervals starting from t_0:

$$\varPhi := \{(u,T) \mid\ t_0 < T \wedge [t_0;T] \subseteq I \wedge \textsf{\textit{is-solution}}_{I:=[t_0;T]}\ u\}$$

For this set, we can show that all solutions u, v in $\varPhi$ take the same values on the intersection of their existence intervals. We do so by falsely assuming that they differ at a point t_1 ($u(t_1) \neq v(t_1)$) and showing that there has to be a maximal point t_m at which u and v are equal. Then, however, one can use the previous theorem about the existence of a unique solution in a neighborhood of t_m, to show that the two solutions have to be equal at larger points than t_m, contradicting its maximality.

One can then define a solution on the interval $J := \bigcup_{(u,T)\in\varPhi}[t_0;T]$ for which *unique-solution*$_{I:=J}$ holds. Additionally, for every other interval K for which there exists a solution, K is a subset of J and the solution is only a restriction. From a mathematical point of view this is an important result, stating the existence of a maximal existence interval for the unique solution:

Theorem 3 (Picard-Lindelöf on an open domain, maximal existence interval)

$$\begin{aligned}&\textsf{\textit{unique-solution}}_{I:=J} \wedge\\ &\forall K \subseteq I.\ \textsf{\textit{is-interval}}\ K \Rightarrow \textsf{\textit{inf}}\ K = t_0 \Rightarrow \textsf{\textit{has-solution}}_{I:=K} \Rightarrow\\ &\qquad (K \subseteq J \wedge (\forall t \in K.\ \textsf{\textit{solution}}_{I:=K}\ t = \textsf{\textit{solution}}_{I:=J}\ t))\end{aligned}$$

5 One-Step Methods

The aim of this paper is to approximate solutions of IVPs with the Euler method. The Euler method is a one-step method: it approximates a function (the solution) in discrete steps, each step operating exclusively on the results of one previous step. For one-step methods in general, one can give assumptions under which the method works correctly – where the error of the approximation goes to zero with the step size.

The methodology is as follows (cf. Bornemann [2]): If the error in one step goes to zero with the step size, the one-step method is called **consistent**. One can show that every consistent one-step method is **convergent**: the global error – the error after a series of steps – goes to zero with the step size, too.

For efficiency reasons, we want to limit the precision of our calculations – which causes rounding errors. The effect of small errors in the execution of a one-step method is studied with the notion of **stability**: The error between the ideal and the perturbed one-step method goes to zero with the step size.

We first give a formal definition of one-step methods, formalize the notions of consistency, convergence and stability. We prove that consistent one-step methods are convergent and stable. We are going to use these definitions and results in the upcoming section to show that the Euler method is consistent and can therefore be used to approximate IVPs.

5.1 Definition

Following the textbook [2], we want to approximate the solution $u : \mathbb{R} \to \mathbb{R}^n$ at discrete values given by $\Delta :: \mathbb{N} \to \mathbb{R}$ with $\forall j.\ \Delta\ j \leq \Delta\ (j+1)$. We notate $\Delta_j := \Delta\ j$, denote by $h_j := \Delta_{j+1} - \Delta_j$ the step size, and by $h_{max} := \mathsf{max}_j\ h_j$ its maximum.

The approximation should be given by a one-step method (or ***g**rid **f**unction*) *gf* such that $\mathit{gf}\ j \approx u\ \Delta_j$. One-step methods use for the approximation at Δ_{j+1} only the information of the previous point at Δ_j. A one-step method *gf* on a grid Δ for a starting value x_0 can therefore be defined recursively. It is characterized by an increment function ψ which gives the slope of the connection (the so called discrete evolution Ψ) between two successive points (depending on the step size h and the position (t, x) of the previous point):

$$h, t :: \mathbb{R}; \quad x, x_0 :: \mathbb{R}^n; \quad \psi, \Psi_\psi :: \mathbb{R} \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n$$

$$\begin{aligned} \Psi_\psi\ h\ t\ x &:= x + h \cdot \psi\ h\ t\ x \\ \mathit{gf}\ \Delta\ \psi\ x_0\ 0 &:= x_0 \\ \mathit{gf}\ \Delta\ \psi\ x_0\ (j+1) &:= \Psi_\psi\ h_j\ \Delta_j\ (\mathit{gf}\ \Delta\ \psi\ x_0\ j) \end{aligned}$$

5.2 Consistency Implies Convergence

We now describe up to which extent one-step methods can be used to approximate an arbitrary function $u : \mathbb{R} \to \mathbb{R}^n$ on an interval $I := [\Delta_0; T]$. We first formalize the notion of consistency (bounding the local error), then summarize a set of required assumptions in the predicate *convergent* from which we show that one-step methods converge.

The error in one step (the local error) is given by $\|u\ (t+h) - \Psi_\psi\ h\ t\ (u\ t)\|$ at a point $(t, u\ t)$ for a step size h. We (as well as the textbook) call a one-step method consistent with u of order p if the local error is in $\mathcal{O}(h^{p+1})$. This means that there exists a constant B such that the local error is less than $B \cdot h^{p+1}$:

$$\begin{aligned}&\textit{consistent}\ u\ B\ p\ \psi := \\ &\quad \big(\forall t \in [\Delta_0; T].\ \forall h \in [0; T-t].\ \|u\ (t+h) - \Psi_\psi\ h\ t\ (u\ t)\| \le B \cdot h^{p+1}\big)\end{aligned}$$

convergent. As in the proof of the Picard-Lindelöf theorem, we need the notion of Lipschitz continuity: The textbook defines a cylindrical neighborhood of radius r around u in which the increment function ψ needs to be Lipschitz continuous. Moreover the increment function is assumed to be consistent with u. The definition of *convergent* summarizes the assumptions required to show convergence.

$$\begin{aligned}\textit{convergent}\ u\ B\ p\ \psi\ r\ L := \textit{consistent}\ u\ B\ p\ \psi \land p > 0 \land B \ge 0 \land L \ge 0 \land \\ \big(\forall t \in [\Delta_0; T].\ \forall h \in [0; T-t]. \\ \textit{lipschitz}\ (\mathcal{B}_r(u\ t))\ (\psi\ h\ t)\ L\big)\end{aligned}$$

We need to give a constant C such that the global error is less than $C \cdot h^p$. This constant depends on B and L and the length S of the interval I. We need this constant as a bound in several upcoming proofs, hence we define it here as $\textit{bound}_S\ B\ L$ for the sake of readability. We want to limit the step size depending on this constant, the order of consistency, and the radius r of the neighborhood with a Lipschitz constant, hence we introduce $\textit{step-bnd}\ B\ L\ p\ r$.

$$\textit{bound}_S\ B\ L := \tfrac{B}{L} \cdot \big(e^{L \cdot S + 1} - 1\big) \qquad \textit{step-bnd}\ B\ L\ p\ r := \sqrt[p]{\frac{\|r\|}{\textit{bound}_{T-\Delta_0}\ B\ L}}$$

Given a one-step method *gf* satisfying the assumptions of *convergent*, we show (inductively on the number of the step j) for a small enough step size that *gf* is convergent: the global error $\|u\ \Delta_j - \textit{gf}\ \Delta\ \psi\ x_0\ j\ \|$ is in $\mathcal{O}(h^p)$.

Theorem 4 (Convergence of One-Step methods)

$$\begin{aligned}&\textit{convergent}\ u\ B\ p\ \psi\ r\ L \Rightarrow h_{max} \le \textit{step-bnd}\ B\ L\ p\ r \Rightarrow \\ &\quad (\forall j.\ \Delta_j \le T \Rightarrow \|u\ \Delta_j - \textit{gf}\ \Delta\ \psi\ x_0\ j\ \| \le \textit{bound}_{T-\Delta_0}\ B\ L \cdot {h_{max}}^p)\end{aligned}$$

5.3 Stability

Since we want to limit the precision of our calculations for reasons of efficiency we need to take the sensitivity against (rounding) errors into account. This is captured by the notion of stability. For a one-step method defined by ψ, we want to study the effect of small perturbations in every step.

For this, we introduce (as in Reinhardt [16]) an error function s and an initial error s_0 and study the perturbed one-step method defined by ψ_s

$$\psi_s\ h\ t\ x := \psi\ h\ t\ x + s\ h\ t\ x$$

stable. Small perturbations do not affect the results of a convergent one-step method too much if we assume a convergent ideal one-step method defined by ψ, a sufficiently small step size, and errors in the order of the step size (the textbook states the theorem for 'sufficiently small' errors, to obtain an explicit bound we basically make the perturbations part of the error we allow for consistency). We summarize this set of assumptions in the definition of *stable*:

$$\begin{aligned} \textit{stable}\ u\ B\ p\ \psi\ r\ L\ s\ s_0 := {} & \textit{convergent}\ u\ B\ p\ \psi\ r\ L\ \wedge \\ & h_{max} \le \textit{step-bnd}\ B\ L\ p\ \frac{r}{2} \wedge s_0 \le \textit{bound}_0\ B\ L \cdot {h_{max}}^p \wedge \\ & (\forall j.\ \|s\ h_j\ \Delta_j\ \ (\textit{gf}\ \Delta\ \psi_s\ (x_0 + s_0)\ j)\| \le B \cdot {h_j}^p) \end{aligned}$$

Under these assumptions, we can show that the error between the distorted $\textit{gf}\ \Delta\ \psi_s\ (x_0 + s_0)$ and the ideal one-step method $\textit{gf}\ \Delta\ \psi\ x_0$ is in $\mathcal{O}(h^p)$:

Theorem 5 (Stability of one-step methods)

$$\begin{aligned} & \textit{stable}\ u\ B\ p\ \psi\ r\ L\ s\ s_0 \Rightarrow \\ & \forall j.\ \Delta_j \le T \Rightarrow \|\textit{gf}\ \Delta\ \psi\ x_0\ j - \textit{gf}\ \Delta\ \psi_s\ (x_0 + s_0)\ j\| \le \textit{bound}_{T-\Delta_0}\ B\ L \cdot {h_{max}}^p \end{aligned}$$

The textbook proof contains an induction quite similar to the one for the proof of convergence, and in fact, we managed to generalize the common part (the accumulation of an error) which we could re-use in both proofs.

6 Euler Method

In this section, we define a simple one-step method, namely the Euler method. We show that the Euler method applied to an IVP is consistent and therefore convergent. For reasons of efficiency, we introduce an approximate implementation of the Euler method on floating point numbers and show that it is stable. We conclude that the approximate Euler method works correctly.

6.1 Definitions

We now assume an IVP *ivp* with domain $I \times D := [\Delta_0; T] \times \mathcal{R}_{b+r}(x_0)$ and take the assumptions from *rect* $T\ (b + r)\ L$. We want to approximate the solution of this IVP with the Euler method on a grid Δ. The Euler method connects a point (t, x) with its subsequent point by a line with the slope given by $f(t, x)$ – independently of the step size. Notice that in contrast to the previous sections, we restrict ourselves and the Euler-method to the univariate case ($f :: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $\mathit{solution} :: \mathbb{R} \to \mathbb{R}$): The proof requires Taylor series expansion, which – in Isabelle/HOL – is only available for the one-dimensional case.

$$\psi^f_{\mathsf{euler}}\ h\ t\ x := f(t, x)$$
$$\mathit{euler}^f\ \Delta\ x_0\ j := \mathit{gf}\ \Delta\ \psi^f_{\mathsf{euler}}\ x_0\ j; \qquad \mathit{euler}^f :: (\mathbb{N} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{N} \to \mathbb{R}$$

6.2 Euler on $\mathbb{R}$

Recall Theorem 4 about the convergence of one step methods and its set of assumptions *convergent*. The Euler method is clearly a one-step method. In order to prove convergence for the Euler method, we need to show that it is Lipschitz continuous and consistent with the solution.

In *rect* $T\ (b + r)\ L$ we have the assumption that f is Lipschitz continuous, hence the Euler method is Lipschitz continuous, too.

We show consistency with the solution with a Taylor series expansion of the *solution* around t, which requires explicit bounds for the derivative of f. Recall that $\|f_c\|$ (as defined in the assumptions of *rect*) is a bound for the values of f on the domain $I \times D$. In *deriv-bnd* we summarize the fact that f' is the (total) derivative of f and that at every point in $I \times D$, the derivative in every possible direction the solution might take (bounded by $\|f_c\|$) is bounded by B. It follows that under these assumptions the Euler method is consistent with the *solution*.

$$\mathit{deriv\text{-}bnd}\ f'\ B := (\forall y \in I \times D.\ \forall dx.\ \|dx\| \le \|f_c\| \Rightarrow f'\ y\ (1, dx) \le 2 \cdot B)$$
$$\mathit{deriv\text{-}bnd}\ f'\ B \Rightarrow \mathit{consistent\ solution}\ B\ 1\ \psi^f_{\mathsf{euler}}$$

The Euler method being consistent and Lipschitz continuous, we can conclude with Theorem 4 that the Euler method converges:

Theorem 6 (Convergence of Euler). *When Picard-Lindelöf guarantees a unique solution on a rectangular domain (rect $T\ (b + r)\ L$) and with explicit bounds on the derivative of f (deriv-bnd $f'\ B$), the Euler method converges for a small enough step size ($h_{max} \le$ step-bnd $B\ L\ 1\ r$) against the solution:*

$$\mathit{convergent\ solution}\ B\ 1\ \psi_{\mathsf{euler}}\ r\ L$$

6.3 Euler on $\mathbb{F}$

We decided to add an implementation of the Euler method on arbitrary-precision floats for efficiency reasons. We define the approximate Euler method $\widetilde{\mathit{euler}}$ as a one-step method operating on floating point numbers. As an increment function, we work with an approximation $\widetilde{f}$ of f in the sense that $(\widetilde{f}\ (\widetilde{t}, \widetilde{x}))_{\mathbb{R}}$ approximates $f((\widetilde{t})_{\mathbb{R}}, (\widetilde{x})_{\mathbb{R}})$ for $\widetilde{t}, \widetilde{x} \in \mathbb{F}$. The error of the approximation may stem from truncating the floating point numbers for reasons of efficiency.

We show that the approximate Euler method works correctly as follows: From Theorem 6, we have a bound on the error between the result of the ideal Euler method and the solution (convergence). We apply Theorem 5 to obtain a bound on the error between the ideal and the approximate Euler method (stability). We summarize the required assumptions in *euler-rounded*: We need *rect* and *deriv-bnd* to show convergence. In addition to that, we need bounds on the error of the approximations $\widetilde{x_0}$ and $\widetilde{f}$ to obtain stability.

$$\begin{aligned}\textit{euler-rounded}\ b\ r\ L\ f'\ B := \ & \textit{rect}\ T\ (b+r)\ L \wedge \textit{deriv-bnd}\ f'\ B\ \wedge \\ & t_0 = (\widetilde{\Delta_0})_{\mathbb{R}} \wedge \|x_0 - (\widetilde{x_0})_{\mathbb{R}}\| \le \textit{bound}_0\ B\ L \cdot (\widetilde{h_0})_{\mathbb{R}}\ \wedge \\ & (\forall j\ \widetilde{x}.\ \|f((\widetilde{\Delta_j})_{\mathbb{R}}, (\widetilde{x})_{\mathbb{R}}) - (\widetilde{f}(\widetilde{\Delta_j}, \widetilde{x}))_{\mathbb{R}}\| \le B \cdot (\widetilde{h_j})_{\mathbb{R}}\end{aligned}$$

One subtle point is the fact that Theorem 5 applies only to one-step methods on real numbers. We therefore need to instantiate the theorem with the perturbed increment function $f_s(t,x) := (\widetilde{f}\ ((t)_{\mathbb{F}}, (x)_{\mathbb{F}}))_{\mathbb{R}}$ and show that the result of $\textit{euler}^{f_s}$ equals $(\widetilde{\textit{euler}^{\widetilde{f}}})_{\mathbb{R}}$, which is easy since $\textit{euler}^{f_s}$ operates exclusively on real numbers representable as floating point numbers.

Having convergence of the ideal Euler method (depending on the step size) and stability of the approximate Euler method (depending on the rounding error), we can approximate IVPs: The execution of $\widetilde{\textit{euler}^{\widetilde{f}}}$ on a grid $\widetilde{\Delta}$ results in an error compared to *solution* that can be made arbitrarily small by choosing smaller step sizes.

Theorem 7 (Convergence of the approximate Euler method on $\mathbb{F}$). *When Picard-Lindelöf guarantees a unique solution on a rectangular domain (*rect $T\ (b+r)\ L$*) and with explicit bounds on the derivative of f and appropriate error bounds for the approximations $(\widetilde{x_0})_{\mathbb{R}}$ and $(\widetilde{f})_{\mathbb{R}}$ (*euler-rounded $b\ r\ L\ f'\ B$*), the approximate Euler method converges for a small enough step size ($\widetilde{h_{max}} \le$ *step-bnd* $B\ L\ 1\ \frac{r}{2}$) against the solution (for every j with $(\widetilde{\Delta_j})_{\mathbb{R}} \le T$):*

$$\left\|\textit{solution}\ (\widetilde{\Delta_j})_{\mathbb{R}} - (\widetilde{\textit{euler}^{\widetilde{f}}}\ \widetilde{\Delta}\ \widetilde{x_0}\ j)_{\mathbb{R}}\right\| \le 2 \cdot \textit{bound}_{T-t_0}\ B\ L \cdot (\widetilde{h_{max}})_{\mathbb{R}}$$

7 Example: $\dot{u}\ t = u^2 - t$

As a simple case-study, we chose the ODE $\dot{u}\ t = (u\ t)^2 - t$ which does not admit a closed formula as solution. In this section we show how we compute $u\ \frac{1}{2}$. First we introduce an IVP depending on the user supplied values $\widetilde{t_0}, \widetilde{x_0}, \widetilde{T}, b, r$:

$$\begin{aligned}f\ (t,x) &:= x^2 - t \\ I \times D &:= [(\widetilde{t_0})_{\mathbb{R}}; (\widetilde{T})_{\mathbb{R}}] \times \mathcal{B}_{b+r}((\widetilde{x_0})_{\mathbb{R}}) \\ (t_0, x_0) &:= ((\widetilde{t_0})_{\mathbb{R}}, (\widetilde{x_0})_{\mathbb{R}})\end{aligned}$$

We then analyze this IVP: We provide the Lipschitz-constant L and the boundary B. We prove a bound of $\|f_c\|$, and the (total) derivative $\dot{f}$ of f:

$$\begin{aligned}L &:= 2 \cdot \max\ \|(\widetilde{x_0})_{\mathbb{R}} - (b+r)\|\ \|(\widetilde{x_0})_{\mathbb{R}} + (b+r)\| \\ B &:= 2 \cdot \max\ \|(\widetilde{x_0})_{\mathbb{R}} - b\|\ \|(\widetilde{x_0})_{\mathbb{R}} + b\| + \tfrac{1}{2} \\ \|f_c\| &\le (\max\ \|(\widetilde{x_0})_{\mathbb{R}} - b\|\ \|(\widetilde{x_0})_{\mathbb{R}} + b\|)^2 \\ \dot{f}\ (t,x)\ (dt, dx) &= 2 \cdot x \cdot dx - dt\end{aligned}$$

The Euler method $\widetilde{\mathit{euler}}^{\tilde{f}}$ is defined with $\tilde{f}(\tilde{t}, \tilde{x}) := \mathit{round}_p(\tilde{x}^2 - \tilde{t})$ on an equidistant grid $\tilde{\Delta}_j := \tilde{t_0} + j \cdot \tilde{h}$. For a fast computation we use the rounding operator $\mathit{round}_p\ x$ which reduces the precision of x, i.e. $\|\mathit{round}_p\ x - x\| \leq 2^{-p}$.

We now set the parameters to

$$\tilde{t_0} := 0,\quad \widetilde{x_0} := 0,\quad b := 1,\quad r := 2^{-8},\quad T := 1,\quad \tilde{h} := 2^{-14},\quad \text{and}\quad p := 14.$$

The error to the solution is bounded by 0.001 due to Theorem 7. We discharge all assumptions with the approximation method [7], as all parameters are fixed.

Theorem 8 (Approximation of the solution to $\dot{u}\ t = u^2 - t$)

$$\forall j \leq 2^{13}.\ \left\| \mathit{solution}\ (\widetilde{\Delta_j})_{\mathbb{R}} - \widetilde{\mathit{euler}}^{\tilde{f}}\ \tilde{\Delta}\ \widetilde{x_0}\ j \right\| \leq 0.001$$

The execution of $\widetilde{\mathit{euler}}^{\tilde{f}}\ \tilde{\Delta}\ \widetilde{x_0}\ 2^{13}$ in the target language SML (where we represent floating point numbers as pairs of integers) returns $-33140952 \cdot 2^{-28} \approx -0.123\ldots$ and takes about 2 seconds on a Core$^{\text{TM}}$2 Duo (E8335) and 2.93 GHz. We put everything together and obtain a result that is correct for two digits:

$$u\ \frac{1}{2} = \mathit{solution}\ (\tilde{\Delta}_{2^{13}})_{\mathbb{R}} \in [-0.124\ldots; -0.122\ldots]$$

This proposition bypassed the LCF kernel of Isabelle since we trust code generation for the approximation method and the evaluation of $\widetilde{\mathit{euler}}^{\tilde{f}}$, but it could (at least in principle) be proved by Isabelle's simplifier.

In order to evaluate the overestimations that stem from the proofs, it is worth noticing that one gets a result inside the same bounds with the choice of 2^{-11} instead of 2^{-14} for stepsize and rounding error. In an experiment with $\dot{u}\ t = u$ (i.e. $u\ t := e^t$), the actual error is more than a factor 22 smaller than the estimated error. Notice also that for our choice of parameters, $\frac{1}{2}$ is the maximum argument where our theorems guarantee correctness.

8 Conclusion and Discussion

We formalized the Picard-Lindelöf theorem to show the existence of a unique *solution* for IVPs in the multivariate case ($\mathbb{R} \to \mathbb{R}^n$). We conducted an analysis of the numerical errors of one-step methods that approximate arbitrary functions in $\mathbb{R} \to \mathbb{R}^n$, which we could use to show that an ideal Euler method euler^f (without rounding errors) approximates the solution (but only in the univariate case, since a multivariate version of Taylor series expansion has not been formalized in Isabelle/HOL yet). Analysis of stability for one-step methods yielded stability for the Euler method: small errors $f - \tilde{f}$ do not affect the global behaviour of an approximate Euler method $\widetilde{\mathit{euler}}^{\tilde{f}}$. See these relations summarized in Fig. 2.

$$\begin{tikzcd}
f \arrow[r, "\text{Picard-Lindelöf}"'] \arrow[d, "="] & \mathit{solution} \arrow[d, "\mathit{convergent}"', "\approx"] \\
f \arrow[r, "\mathit{euler}"'] \arrow[d, "\approx"] & \mathit{euler}^{f} \arrow[d, "\mathit{stable}"', "\approx"] \\
\tilde{f} \arrow[r, "\widetilde{\mathit{euler}}"'] & \widetilde{\mathit{euler}}^{\tilde{f}}
\end{tikzcd}$$

Fig. 2. Relationship between the differential f and the different approximations

$$\begin{tikzcd}
\mathit{rect} \arrow[d] \arrow[rr] & & \mathit{deriv\text{-}bnd} \arrow[d] \arrow[ll] \\
\mathit{euler} \arrow[d, Rightarrow] \arrow[drr, Rightarrow] & & \mathit{euler\text{-}rounded} \arrow[d, Rightarrow] \\
\mathit{consistent} \arrow[d, Rightarrow] & & \mathit{stable} \arrow[dl, Rightarrow] \\
\mathit{convergent} \arrow[r, Rightarrow] & \widetilde{\mathit{euler}}^{\tilde{f}} \approx \mathit{solution} &
\end{tikzcd}$$

Fig. 3. Context hierarchy

Most of the theorems we presented require a large set of assumptions, where the use of locales [4] helped us structuring the theories (compare Fig. 3): We presented the basic Picard-Lindelöf theorem under assumptions with restrictions on the size of the interval *bnd-strip*, then dropped this restriction in *strip*. More realistic applications require restricting the codomain of the solution in *rect* and a variant of the Picard-Lindelöf in the context of open domains of *open-domain* is of mathematical interest. We showed that consistent one step methods (*consistent*) are convergent (*convergent*) and (with additional assumptions) stable (*stable*) and showed these properties for the Euler method. We could conclude that an approximate Euler method converges against the solution.

The Euler method is rarely used in real applications but was relatively easy to analyze. However, the results from one-step methods apply to the widely used Runge-Kutta methods, therefore one can profit from our developments when one implements Runge-Kutta methods of higher order (e.g. the method of Heun or the "classical" Runge-Kutta method) where one only needs to show consistency in order to obtain results about convergence.

In order to obtain explicit bounds for the error of the Euler method in a concrete application, we rely on code generation. The user needs to provide proofs that the concrete application satisfies the assumptions of the contexts in which the error bounds hold. To some extent, an analysis of the differential equation is also necessary when one wants to evaluate the quality of the approximation obtained by some arbitrary numerical method. It might still be desirable to provide more automatization e.g. computing the bounds of the derivative or for deriving a minimum step size automatically.

Our development is available in the AFP [8] and consists of a total of 5020 lines, 1715 for $\overline{\mathcal{C}}$ and the floating-point numbers, 1696 for IVPs, 1217 for one-step methods, and 392 for the example.

References

1. 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)
2. Bornemann, V., Deuflhard, P.: Scientific computing with ordinary differential equations (2002)
3. 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)
4. 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)
5. Harrison, J.: Theorem Proving with the Real Numbers. Ph.D. thesis (1996)
6. Harrison, J.: A HOL Theory of Euclidean Space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
7. Hölzl, J.: Proving inequalities over reals with computation in Isabelle/HOL. In: Reis, G.D., Théry, L. (eds.) Programming Languages for Mechanized Mathematics Systems (ACM SIGSAM PLMMS 2009), pp. 38–45 (2009)
8. Immler, F., Hölzl, J.: Ordinary Differential Equations. Archive of Formal Proofs (April 2012), http://afp.sf.net/entries/Ordinary_Differential_Equations.shtml, Formal proof development
9. Krebbers, R., Spitters, B.: Computer Certified Efficient Exact Reals in COQ. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS (LNAI), vol. 6824, pp. 90–106. Springer, Heidelberg (2011)
10. Krebbers, R., Spitters, B.: Type classes for efficient exact real arithmetic in COQ. CoRR abs/1106.3448 (2011)
11. Melquiond, G.: Proving Bounds on Real-Valued Functions with Computations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 2–17. Springer, Heidelberg (2008)
12. Muñoz, C., Lester, D.R.: Real Number Calculations and Theorem Proving. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 195–210. Springer, Heidelberg (2005)
13. Obua, S.: Flyspeck II: The Basic Linear Programs. Ph.D. thesis, München (2008)
14. O'Connor, R., Spitters, B.: A computer verified, monadic, functional implementation of the integral. Theoretical Computer Science 411(37), 3386–3402 (2010)
15. 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)
16. Reinhardt, H.J.: Numerik gewöhnlicher Differentialgleichungen. de Gruyter (2008)
17. Spitters, B.: Numerical integration in COQ, Mathematics, Algorithms, and Proofs (MAP 2010) (November 2010), www.unirioja.es/dptos/dmc/MAP2010/Slides/Slides/talkSpittersMAP2010.pdf
18. Walter, W.: Ordinary Differential Equations, 1st edn. Springer (1998)

Proof Pearl: A Probabilistic Proof for the Girth-Chromatic Number Theorem

Lars Noschinski

Technische Universität München, Institut für Informatik
noschinl@in.tum.de

Abstract. The Girth-Chromatic number theorem is a theorem from graph theory, stating that graphs with arbitrarily large girth and chromatic number exist. We formalize a probabilistic proof of this theorem in the Isabelle/HOL theorem prover, closely following a standard textbook proof and use this to explore the use of the probabilistic method in a theorem prover.

1 Introduction

A common method to prove the existence of some object is to construct it explicitly. The probabilistic method, which we explain below, is an alternative if an explicit construction is hard. In this paper, we explore whether the use of the probabilistic method is feasible in a modern interactive theorem prover.

Consider the Girth-Chromatic Number theorem from graph theory: Roughly, this states that there exist graphs without short cycles, which nevertheless have a high chromatic number (i.e., one needs a large number of colors to color the vertexes in a way such that no adjacent vertexes have the same color). On first glance, these properties seem contradictory: For a fixed number of vertexes, the complete graph containing all edges has the largest chromatic number. On the other hand, if the cycles are large, such a graph is locally acyclic and hence locally 2-colorable. This discrepancy makes it hard to inductively define a graph satisfying this theorem.

Indeed, the first proof of this theorem given by Erdős [14] used an entirely non-constructive approach: Erdős constructed a probability space containing all graphs of a certain order n. Using tools from probability theory he then proved that, for a large enough n, randomly choosing a graph yields a witness for the Girth-Chromatic Number theorem with a non-zero probability. Hence, such a graph exists. It took 9 more years before a constructive proof was given by Lovász [22].

This use of probability theory is known as *probabilistic method*. Erdős and Rényi are often considered the first conscious users of this method and developed it in their theory of Random Graphs [7, 15]. Other applications include Combinatorics and Number Theory. In this work, we explore how well this technique works in a modern theorem prover.

The well-known Girth-Chromatic Number theorem is one of the early applications of Random Graphs and often given as an example for applications for the probabilistic method. The chromatic number of a graph is the minimal number of colors which is needed to color the vertexes in a way such that adjacent vertexes have different colors. Moreover, the girth g is the size of the shortest cycle in the graph. The Girth-Chromatic number theorem then states that for an arbitrary natural number ℓ there exists a graph G with both $\chi(G) > \ell$ and $g(G) > \ell$.

The proof we present here follows the one given in [11]. The Isabelle/HOL theory files containing our formalization can be found in the Archive of Formal Proofs [26].

The paper is structured as follows: Section 2 provides a brief introduction to Isabelle. Section 3 defines basic graph properties and operations and Section 4 introduces a probability space on graphs. In Section 5 we describe how we handle asymptotic properties. Section 6 gives a high-level description of the proof of the Girth-Chromatic Number theorem before our formalization of this proof is described in Section 7. We reflect on this formalization in Section 8 and review related work in Section 9. Section 10 concludes this paper.

2 Isabelle/HOL

Isabelle/HOL is an implementation of classical Higher Order Logic in the generic interactive theorem prover Isabelle [25]. We just write Isabelle instead of Isabelle/HOL throughout this paper. Formulas and terms are stated in standard mathematical syntax and 2^X denotes the power set of X. The term bound by a quantifier extends as far to the right as possible.

Lists are constructed from nil ($[]$) and cons ($\cdot$) and *hd* and *tl* decompose a list such that $\mathit{hd}(x \cdot xs) = x$ and $\mathit{tl}(x \cdot xs) = xs$.

3 Modeling Graphs

We consider undirected and loop-free graphs $G = (V, E)$ where V and E are sets of vertexes and edges, respectively. Edges are represented as sets of vertexes. For conciseness of presentation, we fix the vertexes to be a subset of $\mathbb{N}$. The graphs may be infinite, however usually we are only interested in finite graphs.

We use V_G and E_G to refer to the vertexes and edges of a graph G. The *order* of a graph is the cardinality of its vertex set. A graph is called *wellformed*, if every edge connects exactly two distinct vertexes of the graph. This is expressed by the following predicate:

$$\mathit{wellformed}(G) := (\forall e \in E_G.\, |e| = 2 \wedge (\forall u \in e.\, u \in V_G))$$

A *walk* is a sequence of vertexes of a graph, such that consecutive vertexes are connected by an edge. We represent walks as non-empty lists of vertexes and define the edge list of a walk recursively. The length of a walk (denoted by $||\cdot||$) is the length (denoted by $|\cdot|$) of its edge list. A cycle is represented as a walk of length at least 3 where first and last vertex are equal, but each other vertex occurs at most once. Note that a cycle of length k is represented by $2k$ different walks.

$$\begin{aligned} \mathit{walk\text{-}edges}([]) &:= [] \\ \mathit{walk\text{-}edges}([x]) &:= [] \\ \mathit{walk\text{-}edges}(x \cdot y \cdot xs) &:= \{x, y\} \cdot \mathit{walk\text{-}edges}(y \cdot xs) \end{aligned}$$

$$||p|| := |\mathit{walk\text{-}edges}(p)|$$

$$\begin{aligned} \mathit{walks}(G) &:= \{p \mid p \neq [] \wedge \mathit{set}(p) \subseteq V_G \wedge \mathit{set}(\mathit{walk\text{-}edges}(p)) \subseteq E_G\} \\ \mathit{cycles}(G) &:= \{p \in \mathit{walks}(G) \mid ||p|| \geq 3 \wedge \mathit{distinct}(\mathit{tl}(p)) \\ &\qquad \wedge \mathit{hd}(p) = \mathit{last}(p)\} \end{aligned}$$

The girth g of a graph is the length of its shortest cycle; the girth of a graph without cycles will be denoted by ∞. As $\mathbb{N}_\infty$, the set of natural numbers extended with ∞, forms a complete lattice, we can define the girth of a graph as the infimum over the length of its cycles:

$$g(G) := \inf_{p \in \mathit{cycles}(G)} ||p||$$

A vertex coloring is a mapping of the vertexes of a graph to some set, such that adjacent vertexes are mapped to distinct elements. We are only interested in the partition defined by this mapping. The chromatic number χ is the size of the smallest such partition.

$$\begin{aligned} \mathit{colorings}(G) := \{C \subseteq 2^{V_G} \mid & \bigcup_{V \in C} = V_G \\ & \wedge (\forall V_1, V_2 \in C.\, V_1 \neq V_2 \Rightarrow V_1 \cap V_2 = \varnothing) \\ & \wedge (\forall V \in C.\, V \neq \varnothing \wedge (\forall u, v \in V.\, \{u, v\} \notin E_G))\} \end{aligned}$$

$$\chi(G) := \inf_{C \in \mathit{colorings}(G)} |C|$$

These definitions suffice to state the Girth-Chromatic Number theorem:

$$\exists G.\ \mathit{wellformed}(G) \wedge \ell < \chi(G) \wedge \ell < g(G)$$

However, we need a few auxiliary definitions; most notably the notion of an independent set and the independence number α.

$$\begin{aligned} E_V &:= \{\{u, v\} \mid u, v \in V \wedge u \neq v\} \\ \mathit{independent\text{-}sets}(G) &:= \{V \mid V \subseteq V_G \wedge E_V \cap E_G = \varnothing\} \\ \alpha(G) &:= \sup_{V \in \mathit{independent\text{-}sets}(G)} |V| \end{aligned}$$

Here, E_V is the set of all (non-loop) edges on V. We also write E_n for $E_{\{1,\ldots,n\}}$.

3.1 Removing Short Cycles

Besides the usual graph theoretic definitions, we will need an operation to remove all short cycles from a graph. For a number k, a short cycle is a cycle with length at most k:

$$\mathit{short\text{-}cycles}(G,k) := \{c \in \mathit{cycles}(G) \mid ||c|| \le k\}$$

We remove the short cycles by repeatedly removing a vertex from a short cycle until no short cycle is left. To remove a vertex from a graph, all edges adjacent to this vertex are also removed.

$$\begin{aligned} G - u &:= (V_G \setminus \{u\}, E_G \setminus \{e \in E_G \mid u \in e\}) \\ \mathit{choose\text{-}v}(G,k) &:= \varepsilon_u(\exists p \in \mathit{short\text{-}cycles}(G,k).\ u \in p) \\ \mathit{kill\text{-}short}(G,k) &:= \mathsf{if}\ \mathit{short\text{-}cycles}(G,k) = \varnothing \\ &\qquad \mathsf{then}\ G\ \mathsf{else}\ \mathit{kill\text{-}short}(G - \mathit{choose\text{-}v}(G,k), k) \end{aligned} \tag{1}$$

To select an arbitrary vertex we use Hilbert's choice operator ε. Given a predicate P, this operator returns either some element satisfying P (if such an element exists) or an arbitrary element from the domain of P otherwise.

Equation (1) defines a recursive function which does not terminate on some infinite graphs. However, an (underspecified) function with this equation can easily defined by the `partial_function` command of Isabelle. To prove some properties about the graphs computed by *kill-short*, a specialized induction rule is useful.

Lemma 1 (Induction rule for *kill-short*). *Let k be a natural number. If for all graphs H both*

$$\mathit{short\text{-}cycles}(H,k) = \varnothing \Rightarrow P(H,k)$$

and

$$\begin{aligned} &\mathit{finite}(\mathit{short\text{-}cycles}(H,k)) \wedge \mathit{short\text{-}cycles}(H,k) \ne \varnothing \\ &\qquad\qquad \wedge P(H - \mathit{choose\text{-}v}(H,k)) \Rightarrow P(H,k) \end{aligned}$$

hold, then $P(G,k)$ holds for all finite *graphs G.*

The canonical induction rule would have $\mathit{finite}(H)$ as assumption for the second rule, but we strengthened the induction hypothesis with the additional assumption $\mathit{finite}(\mathit{short\text{-}cycles}(G,k))$ as it saves a little amount of work when we prove Lemma 4 below. With this induction rule, we can easily prove the following theorems about *kill-short* for finite graphs G:

Lemma 2 (Large Girth). *The girth of $\mathit{kill\text{-}short}(G,k)$ exceeds k, i.e.,*

$$k < g(\mathit{kill\text{-}short}(G,k)) .$$

Lemma 3 (Order of Graph). *$\mathit{kill\text{-}short}(G,k)$ removes at most as many vertexes as there are short cycles, i.e.,*

$$|V_G| - |V_{\mathit{kill\text{-}short}(G,k)}| \le |\mathit{short\text{-}cycles}(G,k)| .$$

Lemma 4 (Independence Number). *Removing the short cycles does not increase the independence number, i.e.,* $\alpha(\textit{kill-short}(G, k)) \leq \alpha(G)$.

Lemma 5 (Wellformedness). *Removing short cycles preserves wellformedness, i.e.,* $\textit{wellformed}(G) \Rightarrow \textit{wellformed}(\textit{kill-short}(G, k))$.

4 Probability Space

There are a number of different probability models which are commonly used for the analysis of random graphs. To prove the Girth-Chromatic number theorem, we consider a series of probability spaces $\mathcal{G}_n$ of graphs of order n, for n going to infinity. $\mathcal{G}_n$ consists of all graphs G with $V_G = \{1, \ldots, n\}$ and $E_G \subseteq E_n$. A randomly chosen graph $G \in \mathcal{G}_n$ contains an edge $e \in E_n$ with probability p_n. As V_G is fixed to $\{1, \ldots, n\}$, a graph $G \in \mathcal{G}_n$ is uniquely defined by its edges; so instead of a space of graphs $\mathcal{G}_n$, we define a space $\mathcal{E}_n$ of edge sets. This turns out to be slightly more convenient.

To define such a probability space in a canonical way, for each edge in E_n one defines a probability space on $\{0, 1\}$, such that 1 occurs with probability p_n and 0 with probability $1 - p_n$. Then, $\mathcal{G}_n$ is identified with the product of these probability spaces.

This construction is supported by Isabelle's extensive library on probability theory [17]. However, the elements of the product space of probability spaces are functions $2^{2^{\mathbb{N}}} \to \{0, 1\}$ which are only specified on E_n. Identifying these with edge sets triggers some amount of friction in a theorem prover. To avoid this, we construct a probability space on edge sets without using the product construction. This is easily possible as $\mathcal{E}_n$ is finite for all n.

For the definition of $\mathcal{E}_n$, we consider the following. In the setting above, the probability that a randomly chosen edge set contains a fixed edge e is p_n; the probability of the negation is $1 - p_n$. As the probabilities of the edges are independent, the probability that a randomly chosen edge set is equal to a fixed set $E \subseteq E_n$ is the product of the edge probabilities, i.e., $p_n^{|E|} \cdot (1 - p_n)^{|E_n - E|}$.

Definition 6 (Probability Space on Edges). *Let* $n \in \mathbb{N}$ *and* $p \in \mathbb{R}$ *with* $0 \leq p \leq 1$. *Let* $f(E) = p^{|E|} \cdot (1 - p)^{|E_n - E|}$ *for all* $E \in 2^{E_n}$. *Then* $\mathcal{E}_{n,p} = (2^{E_n}, \mathcal{P}_{n,p})$ *is the probability space whose domain consists of all the subsets of* E_n *and whose probability function is* $\mathcal{P}_{n,p}(X) = \Sigma_{E \in X} f(E)$ *for all* $X \subseteq E_n$. *When a function* $p : \mathbb{N} \to \mathbb{R}$ *is given from the context, we also write* $\mathcal{E}_n$ *and* $\mathcal{P}_n$ *instead of* $\mathcal{E}_{n,p_n}$ *and* $\mathcal{P}_{n,p_n}$.

Isabelle's probability library provides a *locale* [4] for probability spaces. One option to specify such a space is by a finite domain X and a function f with the following properties: For each $x \in X$ holds $0 \leq f(x)$ and we have $\sum_{x \in X} f(x) = 1$. When we show that those two properties hold in Def. 6, then Isabelle's locale mechanism transfers all lemmas about probability spaces to $\mathcal{E}_{n,p}$. We need in particular the following lemma:

Lemma 7 (Markov's Inequality). *Let $P = (X, \mu)$ be a probability space, $c \in \mathbb{R}$ and $f : X \to \mathbb{R}$ such that $0 < c$ and for all $x \in X$ holds $0 \le f(x)$. Then*

$$\mu(\{x \in X \mid c \le f(x)\} \le 1/c \cdot \Sigma_{x \in X}(f(x) \cdot \mu\{x\}) \ .$$

To prove that $\mathcal{E}_{n,p}$ is an instance of the locale of probability spaces we need the lemma below.

Lemma 8 (Sum of Probabilities Equals 1). *Let S be a finite set. Then for all $p \in \mathbb{R}$ holds $\Sigma_{A \subseteq S} \left(p^{|A|} \cdot (1-p)^{|S-A|}\right) = 1$.*

A similar lemma describes the probability of certain sets of edge sets.

Lemma 9 (Probability of Cylinder Sets). *Let $\mathcal{E}_{n,p}$ be a probability space and $cyl_n(A, B) := \{E \subseteq E_n \mid (\forall x \in A.\ x \in E) \wedge (\forall x \in B.\ x \notin E)\}$ the set of all edge sets containing A but not B. Then $\mathcal{P}_{n,p}(cyl_n(A, B)) = p^{|A|} \cdot (1-p)^{|B|}$ for all disjoint $A, B \subseteq E_n$.*

5 Handling Asymptotics

As mentioned in Section 4, we consider a series of probability spaces, as the order grows towards infinity. In many cases, it suffices if a property P holds after some finite prefix, i.e., $\exists k.\ \forall n > k.\ P(n)$. Often, we can avoid dealing with these quantifiers directly. For example, to prove

$$(\exists k_1.\ \forall n > k_1.\ P(n)) \wedge (\exists k_2.\ \forall n > k_2.\ Q(n)) \Rightarrow \exists k_3.\ \forall n > k_3.\ R(n)$$

we can prove $\exists k.\ \forall n > k.\ P(n) \wedge Q(n) \Rightarrow R(n)$ or even just $\forall n.\ P(n) \wedge Q(n) \Rightarrow R(n)$ instead. However, such a rule would be inconvenient to use in practice, as proof automation tends to destroy the special form of the quantifiers. This can be prevented by using a specialized constant instead of the quantifiers. In Isabelle, such a constant (with suitable lemmas) is already available in the form of *filters* [8] and the *eventually* predicate. Filters generalize the concept of a sequence and are used in topology and analysis to define a general notion of convergence; they can also be used to express quantifiers [6]. In rough terms, a filter is a non-empty set of predicates closed under conjunction and implication and *eventually* is the membership test. We use *eventually* with the filter

$$\textit{sequentially} := \{P \mid \exists k.\ \forall n > k.\ P(n)\}$$

as kind of a universal quantifier. This fits nicely Isabelle's definition of a limit:

$$\lim_{n \to \infty} f(n) = c \Rightarrow \forall \varepsilon.\ \textit{eventually}\,((\lambda n.\ |f(n) - c| < \varepsilon), \textit{sequentially})$$

The formula $\exists k.\ \forall n > k.\ P(n)$ is equivalent to *eventually*$(P, \textit{sequentially})$. We will denote this as $\forall^\infty n.\ P(n)$ or write "$P(n)$ holds for large n". We mostly used the following three rules when dealing with this quantifier:

$$\frac{\forall n.\ k < n \Rightarrow P(n)}{\forall^\infty n.\ P(n)} \qquad (\textit{eventually-sequentiallyI})$$

$$\frac{\forall^\infty n.\ P(n) \quad \forall^\infty n.\ (P(n) \Rightarrow Q(n))}{\forall^\infty n.\ Q(n)} \qquad (\textit{eventually-rev-mp})$$

$$\frac{\forall^\infty n.\ P(n) \quad \forall^\infty n.\ Q(n) \quad \forall n.\ (P(n) \wedge Q(n)) \Rightarrow R(n)}{\forall^\infty n.\ R(n)} \qquad (\textit{eventually-elim2})$$

Apart from rule *eventually-sequentiallyI*, these hold for the *eventually* predicate in general. The rule *eventually-elim2* is actually just a convenience rule, which can be easily derived from the other two rules by dropping the condition $k < n$.

6 Proof Outline

We start with a high-level outline of the proof. Let ℓ be a natural number. A cycle c with $||c|| \leq \ell$ is called a *short cycle*. We recall the Girth-Chromatic Number theorem:

$$\exists G.\ \textit{wellformed}(G) \wedge \ell < \chi(G) \wedge \ell < g(G)$$

Instead of working with the chromatic number, we will work with the independence number α. Estimating probabilities for this number is easier, as an independent set is a cylinder set, cf. Lemma 9. The following lemma relates chromatic and independence number.

Lemma 10 (Lower Bound for $\chi(G)$). *For all graphs G, $|G|/\alpha(G) \leq \chi(G)$.*

The basic idea of the probabilistic proof of existence is to show that, for large enough n, choosing a random graph $G \in \mathcal{G}_n$ (respectively $E \in \mathcal{E}_n$) yields a graph with the desired properties with a non-zero probability.

A reasonable approach would be to choose the probability function p_n, such that we can show $\mathcal{P}_n\{G \mid g(G) < \ell\} + \mathcal{P}_n\{G \mid \alpha(G) > n/\ell\} < 1$. This would imply that a graph G satisfying neither $g(G) < \ell$ nor $\chi(G) < \ell$ exists, i.e., a graph satisfying the Girth-Chromatic number property. However, such a probability does not exist [11]. Instead, by choosing p_n correctly, we can show the weaker property

$$\mathcal{P}_n\{G \mid n/2 \leq |\textit{short-cycles}(G, \ell)|\} + \mathcal{P}_n\{G \mid 1/2 \cdot n/\ell \leq \alpha(G)\} < 1$$

and obtain a graph with at most $n/2$ short cycles and an independence number less than $1/2 \cdot n/\ell$ (i.e., $2\ell < \chi(G)$ by Lemma 10). From this graph, we remove a vertex from every short cycle. The resulting graph then has large girth and the chromatic number is still large.

7 The Proof

As a first step, we derive an upper bound for the probability that a graph has at least $1/2 \cdot n/k$ independent vertexes.

Lemma 11 (Probability for many Independent Edges). *Given $n, k \in \mathbb{N}$ such that $2 \le k \le n$, we have*

$$\mathcal{P}_n\{E \subseteq E_n \mid k \le \alpha(G_{n,E})\} \le \binom{n}{k}(1-p_n)^{\binom{k}{2}} \ .$$

Proof. Holds by a simple combinatorial argument and Lemma 9.

Lemma 12 (Almost never many Independent Edges). *Assume that $0 < k$ and $\forall^\infty n.\ 0 < p_n \wedge p_n < 1$. If in addition $\forall^\infty n.\ 6k \cdot \ln n/n \le p_n$ holds, then there are almost never more then $1/2 \cdot n/k$ independent vertexes in a graph, i.e.,*

$$\lim_{n\to\infty} \mathcal{P}_n\{E \subseteq E_n \mid 1/2 \cdot n/k \le \alpha(G_{n,E})\} = 0$$

Proof. With Lemma 11.

Then we compute the expected number of representatives of cycles of length k in a graph. Together with Markov's Lemma, this will provide an upper bound of $\mathcal{P}_n\{E \in \mathcal{E}_n \mid n/2 \le |\mathit{short\text{-}cycles}(G_{n,E}, \ell)|\}$.

Lemma 13 (Mean Number of k-Cycles). *If $3 \le k < n$, then the expected number of paths of length k describing a cycle is*

$$\left(\Sigma_{E\in\mathcal{E}_n}|\{c \in \mathit{cycles}(G_{n,E}) \mid k = ||c||\}| \cdot \mathcal{P}_n(\{E\})\right) = \frac{n!}{(n-k)!} \cdot p^k$$

We arrive at our final theorem:

Theorem 14 (Girth-Chromatic Number). *Let ℓ be a natural number. Then there exists a (wellformed) graph G, such that $\ell < g(G)$ and $\ell < \chi(G)$:*

$$\exists G.\ \mathit{wellformed}(G) \wedge \ell < g(G) \wedge \ell < \chi(G)$$

To prove this, we fix $p_n = n^{\varepsilon - 1}$ where $\varepsilon = 1/(2\ell)$ and assume without loss of generality that $3 \le \ell$. These assumptions hold for all of the following propositions. With Lemma 13, we can derive an upper bound for the probability that a random graph of size n has more than $n/2$ short cycles:

Proposition 15

$$\forall^\infty n.\ \mathcal{P}_n\{E \subseteq E_n \mid n/2 \le |\mathit{short\text{-}cycles}(G_{n,E}, \ell)|\} \le 2(\ell-2)n^{\varepsilon\ell - 1}$$

As this converges to 0 for n to infinity, eventually the probability will be less than 1/2:

Proposition 16

$$\forall^\infty n.\ \mathcal{P}_n\{E \subseteq E_n \mid n/2 \le |\mathit{short\text{-}cycles}(G_{n,E}, \ell)|\} < 1/2$$

Similarly, with these choices, the conditions of Lemma 12 are satisfied:

Proposition 17

$$\forall^\infty n.\ \mathcal{P}_n\{E \subseteq E_n \mid 1/2 \cdot n/\ell \leq \alpha(G_{n,E})\} < 1/2$$

Therefore, the sum of these probabilities will eventually be smaller than 1 and hence, with a non-zero probability, there exists a graph with only few short cycles and a small independence number:

Proposition 18. *There exists a graph $G \in \mathcal{G}_n$ satisfying both $1/2 \cdot n/\ell > \alpha(G)$ and $n/2 > |\mathit{short\text{-}cycles}(G, \ell)|$.*

By removing the short cycles, this graph will be turned into a witness for the Girth-Chromatic Number theorem. This completes the proof of Theorem 14.

Proposition 19. *Let G be a graph obtained from Lemma 18. Then the graph $H := \mathit{kill\text{-}short}(G, \ell)$ satisfies $\ell < g(H)$ and $\ell < \chi(H)$. Moreover, H is well-formed.*

Proof. By Lemmas 2–5 and 10.

Actually, we almost proved an even stronger property: The probabilities in Propositions 16 and 17 converge both to 0, so almost all graphs satisfy the condition of Proposition 18. Hence, almost every graph can be turned into a witness for the Girth-Chromatic Number theorem by removing the short cycles. This is typical for many proofs involving the probabilistic method.

8 Discussion

In this work, we formally proved the Girth-Chromatic Number theorem from graph theory, closely following the text book proof. The whole proof consists of just 84 theorems (1439 lines of Isabelle theories), split into three files and is therefore quite concise. Around 41 of these lemmas are of general interest, reasoning about reals with infinity and some combinatorial results. Partly, these have been added to the current developer version of Isabelle. Moreover, 18 lemmas are given about basic graph theory and the core proof of the theorem consists of the remaining 25 lemmas (around 740 lines). For the core proof, we mostly kept the structure of the text book proof, so auxiliary propositions only needed for one lemma are not counted separately.

The result looks straight-forward, but there are some design choices we like to discuss. In an early version of this formalization, we represented edges by an explicit type of two-element sets. However, it turned out that this made some proof steps a lot more complicated: Isabelle does not support subtyping, so defining a two-element-set type yields a new type disjoint from sets with a partial type constructor. When we need to refer to the vertexes connected by an edge, this partiality makes reasoning harder. This easily offsets the little gain (we only need wellformedness explicitly in two theorems) an explicit edge type gives in our setting.

One should note that our definition of the chromatic number is not as obviously correct as it appears from the first glance: For an infinite graph G, $\chi(G) = 0$. This is due to the standard definition of cardinality in Isabelle mapping infinite sets to 0. We decided not to care about this, as we only are interested in finite graphs (and our final theorem assures a positive chromatic number anyway).

The main reason we decided to use $\mathbb{N}_\infty$ instead of $\mathbb{N}$ was to be able to give a natural definition of the girth – without infinity, we would need an extra predicate to handle the "no cycles" case. A nice side effect is that α and χ are easier to handle, as we do not have to care about emptyness or finiteness to evaluate infimum and supremum. However, as a result of this choice, we need to deal with real numbers including infinity ($\mathbb{R}_\infty$, with $\pm\infty$). If this had not been already available as a library, it would probably have been easier to avoid infinity altogether and special case the girth of acyclic graphs.

Our use of *eventually* turned out to be quite rewarding. For the proofs for Lemma 12 and the propositions for Theorem 14 we quite often collect a number of facts holding for large n and eliminate them like in Section 5. This made for more elegant proofs, as we needed less bookkeeping for (mostly irrelevant) explicit lower bounds.

Now, which capabilities are needed to use the probabilistic method in a theorem prover? Obviously some amount of probability theory. Different fragments of probability theory are now formalized in many theorem provers, including HOL4, HOL-light, PVS, Mizar and Isabelle [12,17,18,21,23]. Surprisingly, for the proof presented here, not much more than Markov's Inequality is required. For other proofs, more stochastic vocabulary (like variance and independence) is needed.

If one makes the step from finite to infinite graphs (for example to prove the Erdős-Rényi theorem that almost all countably infinite graphs are isomorphic [13,27]), infinite products of probability spaces are required. To our knowledge, the only formalization of these is found in Isabelle [17].

Furthermore, good support for real arithmetic including powers, logarithms and limits is needed. Isabelle has this, but proving inequalities on complex terms remains tedious as often only very small proof steps are possible. However, the calculational proof style [5] (inspired by Mizar) is very helpful here.

In the future, an automated reasoner for inequalities over real-value functions like MetiTarski [1] might be useful. However, the set of a few example inequalities from our proof which L. Paulson kindly tested for us is still outside the reach of MetiTarski.

9 Related Work

Proofs with the probabilistic method often lead to randomized algorithms. Probably the first formalization in this area is Hurd's formalization of the Miller-Rabin primality test [19]; other work on this topic is available in Coq [3]. A constructive proof of a theorem similar to the Girth-Chromatic Number theorem was formalized by Rudnicki and Stewart in Mizar [28].

There are a few general formalizations of graph theory available in various theorem provers, for example [9, 10, 20]; but often proof developments rather use specialized formalizations of certain aspects of graph theory [16, 24] to ease the proof. For the Girth-Chromatic Number theorem, the common definition of graphs as pairs of vertexes and edges seems quite optimal. Even though, we did not find the lack of a general graph theory in Isabelle to be a major obstacle: The Girth-Chromatic Number theorem does not rely on any deep properties about graphs and the formalization of graphs we give here is rather straight-forward.

10 Conclusion

We gave a concise (and, to our knowledge, the first) formal proof for the well-known Girth-Chromatic Number theorem and explored the use of the probabilistic method in theorem provers, which worked well for this theorem. It will be interesting to see whether this continues to hold true for more involved theorems. An interesting example for this could be Lovász Local Lemma: Many probabilistic proofs show not only that the probability is non-zero, but even that it tends to 1 for large graphs. The Local Lemma can be used to show that a property holds with a positive, but very small probability. This enables some combinatorical results, for which no proof not involving this lemma is known [2].

Acknowledgments. We thank the anonymous reviewers for their feedback and pointing us to [10].

References

1. Akbarpour, B., Paulson, L.C.: Metitarski: An automatic theorem prover for real-valued special functions. JAR 44, 175–205 (2010)
2. Alon, N., Spencer, J.H.: The Probabilistic Method. Wiley (2000)
3. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in COQ. Science of Computer Programming 74(8), 568–589 (2009)
4. Ballarin, C.: Interpretation of Locales in Isabelle: Theories and Proof Contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 31–43. Springer, Heidelberg (2006)
5. Bauer, G., Wenzel, M.T.: Calculational Reasoning Revisited (An Isabelle/Isar Experience). In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 75–90. Springer, Heidelberg (2001)
6. van Benthem, J.F.A.K., ter Meulen, A.G.: Generalized quantifiers in natural language. de Gruyter (1985)
7. Bollobás, B.: Random Graphs. Academic Press (1985)
8. Bourbaki, N.: General Topology (Part I). Addison-Wesley (1966)
9. Butler, R.W., Sjogren, J.A.: A PVS graph theory library. Tech. rep., NASA Langley (1998)
10. Chou, C.-T.: A Formal Theory of Undirected Graphs in Higher-Order Logic. In: Melham, T.F., Camilleri, J. (eds.) HUG 1994. LNCS, vol. 859, pp. 144–157. Springer, Heidelberg (1994)

11. Diestel, R.: Graph Theory, GTM, 4th edn., vol. 173. Springer (2010)
12. Endou, N., Narita, K., Shidama, Y.: The lebesgue monotone convergence theorem. Formalized Mathematics 16(2), 171–179 (2008)
13. Erdős, P., Rényi, A.: Asymmetric graphs. Acta Mathematica Hungarica 14, 295–315 (1963)
14. Erdős, P.: Graph theory and probability. Canad. J. Math. 11(11), 34–38 (1959)
15. Erdős, P., Rényi, A.: On random graphs I. Publ. Math. Debrecen. 6, 290–297 (1959)
16. Gonthier, G.: A computer-checked proof of the Four Colour Theorem (2005)
17. 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)
18. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)
19. Hurd, J.: Verification of the Miller-Rabin probabilistic primality test. JLAP 50(1-2), 3–21 (2003)
20. Lee, G., Rudnicki, P.: Alternative graph structures. Formalized Mathematics 13(2), 235–252 (2005), Formal Proof Development
21. Lester, D.R.: Topology in PVS: continuous mathematics with applications. In: Proc. AFM, pp. 11–20. ACM (2007)
22. Lovász, L.: On chromatic number of finite set-systems. Acta Mathematica Hungarica 19, 59–67 (1968)
23. 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)
24. Nipkow, T., Bauer, G., Schultz, P.: Flyspeck I: Tame Graphs. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 21–35. Springer, Heidelberg (2006)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002), `http://isabelle.in.tum.de/dist/Isabelle2011-1/doc/tutorial.pdf`
26. Noschinski, L.: A probabilistic proof of the girth-chromatic number theorem. In: The Archive of Formal Proofs (February 2012), `http://afp.sf.net/entries/Girth_Chromatic.shtml`, Formal Proof Development
27. Rado, R.: Universal graphs and universal functions. Acta Arithmetica 9, 331–340 (1964)
28. Rudnicki, P., Stewart, L.: The Mycielskian of a graph. Formalized Mathematics 19(1), 27–34 (2011)

Standalone Tactics Using OpenTheory

Ramana Kumar[1,⋆] and Joe Hurd[2]

[1] University of Cambridge
Ramana.Kumar@cl.cam.ac.uk
[2] Galois, Inc.
joe@gilith.com

Abstract. Proof tools in interactive theorem provers are usually developed within and tied to a specific system, which leads to a duplication of effort to make the functionality available in different systems. Many verification projects would benefit from access to proof tools developed in other systems. Using OpenTheory as a language for communicating between systems, we show how to turn a proof tool implemented for one system into a standalone tactic available to many systems via the internet. This enables, for example, LCF-style proof reconstruction efforts to be shared by users of different interactive theorem provers and removes the need for each user to install the external tool being integrated.

1 Introduction

There are many LCF-style systems for interactively developing machine-checked formal theories, including HOL4 [1], HOL Light [2], ProofPower [3] and Isabelle/HOL [4]. The logic implemented by these systems is essentially the same, but the collections of theory libraries and proof tools built on top of the logical kernels differ. Where similar proof tools exist in multiple systems it is usually the result of duplicated effort.

Examples of duplicated effort on tactics include the integration of external tools into HOL-based provers. For instance, Kumar and Weber [5] and Kunčar [6] give independent integrations of a quantified boolean formula solver into HOL4 and HOL Light. Weber and Amjad [7] give high-performance integrations of SAT solvers into three HOL-based systems; each integration requires a separate implementation. Sledgehammer [8] is only available for Isabelle/HOL, but its functionality would also be useful in other systems.

In addition to the development effort, the cost of maintenance can also be multiplied over different systems, and improvements in functionality can become restricted to a single system unnecessarily. For instance, the Metis first order logic prover [9] is integrated in multiple systems in the HOL family, but the HOL4 version is very old compared to the latest version in Isabelle/HOL. Slind's TFL package for defining recursive functions [10], originally implemented for both Isabelle/HOL and HOL4, was superseded in Isabelle/HOL by Krauss's function

⋆ Supported by the Gates Cambridge Trust.

definition package [11]. The improvements of Krauss's method over TFL ought to be applicable to other HOL-based provers, but a direct reimplementation would require substantial effort.

It makes sense to speak of similar proof tools in different interactive theorem provers not just because they implement essentially the same logic, but also because there is a shared base of concepts: booleans, inductive datatypes, recursive functions, natural numbers, lists, sets, etc. The OpenTheory standard library [12] formalises this shared base as a collection of theory packages containing proofs written in the simple *article* format designed for storing and sharing higher order logic theories [13]. We use OpenTheory as a language for interactive theorem provers to communicate with proof tools on a remote server, and thereby obtain the following two benefits:

1. **Proof Articles:** A standard format to encode the goals that will be sent to the remote proof tools and the proofs that will be received in response.
2. **Standard Library:** An extensible way to fix the meaning of constants and type definitions between systems.

We contend that proof tools for interactive theorem provers need only be written and maintained in one place rather than once per system, using *standalone tactics* that are available online and communicate using OpenTheory. An added advantage when the tactic is an integration of an external tool is that a user of the interactive theorem prover need not install the external tool: it only needs to be available on the server hosting the standalone tactic.

The contributions of this rough diamond are:

1. A general method for turning existing proof tools implemented in interactive theorem provers into standalone tactics (Section 2).
2. Preliminary results profiling the performance of working examples of standalone tactics (Section 3).

2 Lifting Proof Tools into the Cloud

2.1 OpenTheory for Tactic Communication

An example: the user of an interactive theorem prover faced with the goal

$$\forall n.\ 8 \leq n \Rightarrow \exists s, t.\ n = 3s + 5t$$

decides to pass it off to a standalone tactic for linear arithmetic.

The input for the standalone tactic is the goal term, and the output is a proof of the theorem. Standalone tactics use the OpenTheory article format for communicating terms and proofs. The interactive theorem prover serializes the goal term from its local internal format to an article file, and sends the article over the internet to the standalone tactic. If successful, the standalone tactic sends back another article file encoding a proof of the goal, which the interactive theorem prover replays through its logical kernel to create the desired theorem.

This example illustrates the key requirements for an interactive theorem prover to use standalone tactics:

1. Ability to replay proofs by reading OpenTheory articles.
2. Ability to write terms as OpenTheory articles.
3. Ability to communicate with external programs.

Requirements 1 and 2 can be satisfied for an interactive theorem prover by implementing an OpenTheory interface that can interpret and construct articles. The central concept in OpenTheory is that of a *theory package*, $\Gamma \rhd \Delta$, which proves that the set of theorems Δ logically derive from the set of assumptions Γ. An article is a concrete representation of a theory package, consisting of instructions for a virtual machine whose operations include construction of types and terms, and the primitive inference rules of higher order logic. To *read* an article, an interactive theorem prover performs the primitive inferences and other instructions listed in the file. The OpenTheory logical kernel is based on HOL Light's logical kernel, and the instructions are chosen to make it easy to read articles into any system that can prove theorems of higher order logic.

An article file represents a theory $\Gamma \rhd \Delta$. By taking Δ to be the set of theorems proved by a proof tool and Γ to be the set of theorems used by the proof tool, we can view the result of executing a proof tool as a logical theory. In our example above of using a linear arithmetic standalone tactic on the given goal, this theory might be

$$\left\{\begin{array}{l} \vdash \forall n.\ n + 0 = n \\ \vdash \forall m, n.\ mn = nm \\ \cdots \end{array}\right\} \rhd \left\{ \vdash \forall n.\ 8 \leq n \Rightarrow \exists s, t.\ n = 3s + 5t \right\}$$

where the assumptions consist of a collection of standard arithmetic facts.

The main benefit of using OpenTheory for communication is that it provides a standard ontology for fixing the meanings of constants and type operators between different systems. For example, the numerals 3, 5 and 8 in the example goal term can be encoded in binary using the standard constants `Number.Natural.bit0` and `Number.Natural.bit1`. The full names and properties of these constants are indicated in the OpenTheory standard library, and interactive theorem provers can maintain translations to and from their local names and theorems. A system using a different encoding for numbers (say unary, with `Number.Natural.suc` and `Number.Natural.zero`) could use additional standalone tactics to translate between encodings.

Implementing an OpenTheory interface to satisfy Requirements 1 and 2 above carries the additional benefit of giving the interactive theorem prover access to all logical theories stored as OpenTheory packages, not just those that are the output of standalone tactics.

2.2 Extracting Tactics from Interactive Theorem Provers

There are two approaches to obtaining a standalone tactic: either write one directly; or extract an existing tactic from an interactive theorem prover. We have experimented with the second approach, extracting tactics from HOL4 and from HOL Light. The procedure is reasonably lightweight, but less flexible than writing a standalone tactic directly. For tactic extraction to succeed, the key requirements on the interactive theorem prover are:

1. Ability to read and write OpenTheory article files.
2. Ability to record proofs and reduce them to the OpenTheory kernel.
3. Ability to make a standalone executable encompassing the tactic functionality separated from the usual interface to the interactive theorem prover.

Just as the requirements for an interactive theorem prover to use standalone tactics also enable it to import OpenTheory packages in general, the first two requirements to create standalone tactics also enable a system to create and export OpenTheory packages. (The last requirement enables running on a server.)

Requirement 2 poses the highest barrier: a standalone tactic must record each proof at a level of detail sufficient to prove the same theorem using the OpenTheory kernel. The following method can be used if the system implementing the tactic has an LCF-style design, that is, theorems can only be created by a small number of primitive inference rules: (i) augment the internal theorem type with a type of proofs to track the system primitive inferences used; and (ii) express each system primitive inference as a derived rule of the OpenTheory kernel. We applied this method to meet Requirement 2 for both HOL4 and HOL Light. For some primitive inferences (such as reflexivity of equality) there is a direct translation to the OpenTheory logical kernel. But, for example, HOL4's primitive rule for definition by specification must be emulated in the OpenTheory kernel, for example by using Hilbert choice. Although Isabelle/HOL uses the LCF architecture, its logical kernel is quite different from OpenTheory; we therefore expect translating Isabelle/HOL proofs to be more difficult than HOL4 and HOL Light proofs.

To satisfy Requirement 3 in HOL4, we used the 'export' facility of the Poly/ML compiler, which creates a standalone executable that runs an ML function. For each tactic, we captured a function that reads an article representing the input term, runs the tactic (recording the proof), and writes an article containing the proof. The situation for HOL Light is more complicated because OCaml does not provide such an 'export' facility, and a HOL Light session typically starts by proving the standard library. We used a general-purpose checkpointing facility to capture a HOL Light session at the point where it is ready to read an article and run a tactic.

3 Preliminary Performance Results

We collected preliminary performance data for two test standalone tactics extracted from HOL4, and called from HOL Light. They are QBF [5], which proves quantified boolean formulas, and SKICo, which rewrites terms to combinatory form like so: $\vdash (\forall x.\ x \lor \neg x) = (\forall)\ (\mathtt{S}\ (\lor)\ (\neg))$. We used the following three test goals:

Table 1. Performance profiling for the test standalone tactics

Tactic-Problem	**Goal Size (b)**	**Goal Time (s)**	**Remote Time (s)**	**Proof Size (b)**	**Proof Time (s)**	**Total Time (s)**	**Local Time (s)**
QBF-1	927	0.001	1.064	10,991	0.022	1.088	0.002
QBF-2	1,474	0.001	1.892	79,944	0.139	2.034	0.024
QBF-3	1,546	0.001	1.821	91,639	0.172	1.996	0.024
SKICo-1	927	0.001	1.212	20,047	0.041	1.255	0.000
SKICo-2	1,474	0.002	1.557	52,249	0.113	1.673	0.001
SKICo-3	1,546	0.002	1.716	60,642	0.125	1.844	0.005

1. $\forall x.\ x \lor \neg x$
2. $\exists p.\ (\forall q.\ p \lor \neg q) \land \forall q.\ \exists r.\ r$
3. $\exists x.\ \forall y.\ \exists z.\ (\neg x \lor \neg y) \land (\neg z \lor \neg y)$

For each invocation of a standalone tactic on a test goal, Table 1 profiles the time and space requirements of the three phases of execution: encoding the goal as an article; communicating with and executing the standalone tactic remotely; and replaying the proof article received. For comparison, the time to run the tactic locally within HOL4 is given in the rightmost column.

The sizes of the articles for the test goals and the resulting proofs are comparable to the typical size of web requests and the resulting pages, so we can be confident that we are within the normal operating range of the web tools we use (`curl` on the client and CGI scripts on the server). For problems involving larger articles (bigger goals or longer proofs) we may wish to compress them using `gzip` before sending them over the network—previous experiments showed a compression ratio of 90% is typical for article files [13].

Turning now to execution time, we can see that it is significantly more expensive to call a standalone tactic over the internet compared to executing it locally. However, most of the time is spent on 'Remote Time', which includes communicating with the remote standalone tactic and waiting for it to read the goal article, run the proof tool, and write the resulting proof article. Using `traceroute` we see a 0.173s network delay between the test client in Portland, OR, USA and the test server in Cambridge, UK, which accounts for at least 0.346s of delay. The overall time is in the 1–2s range, which is very slow for workaday tactics but may well be tolerated by the user to gain access to the functionality of a proof tool on another system.

4 Related Work

The PROSPER project [14] pioneered the technique of sending goals over the internet to a remote solver, and packaging such procedures as tactics in the HOL4 theorem prover. The standalone tactics described in this paper further systematize this by using OpenTheory as a standard language and ontology to make it easier for interactive theorem provers and remote solvers to communicate.

An impressive example of providing reasoning infrastructure over the internet is System on TPTP [15], which enables a user to remotely execute a collection of automatic theorem provers on a problem expressed in a standard format. The usual scenario is to decide the validity of first order logic formulas (TPTP format), but there is also support for higher order terms (THF format), and for returning proofs expressed in a standard language (TSTP format).

The idea of separate reasoning tools communicating to enhance a proof development environment is also being pursued in the Evidential Tool Bus [16] and the MathServe System [17]. This idea is a natural extension of the integration of automatic tools with interactive theorem provers.

5 Conclusion

We have shown how, using OpenTheory for communication, we can write tools for higher order logic reasoning tasks as standalone tactics, making them available to multiple interactive theorem provers and independently maintainable. Existing proof tools can be extracted from their home systems for reuse. There is a substantial but hopefully tolerable overhead of communicating goals and proofs in article files over the internet.

References

1. Slind, K., Norrish, M.: A brief overview of HOL4. In: [18], pp. 28–32
2. 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)
3. Arthan, R.: ProofPower manuals (2004), http://lemma-one.com/ProofPower
4. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: [18], pp. 33–38
5. Kumar, R., Weber, T.: Validating QBF validity in HOL4. In: [19], pp. 168–183
6. Kunčar, O.: Proving valid quantified boolean formulas in HOL Light. In: [19], pp. 184–199
7. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2009)
8. 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)
9. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Vito, B.D., Muñoz, C. (eds.) STRATA 2003, Number NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68 (September 2003)
10. Slind, K.: Reasoning about Terminating Functional Programs. PhD thesis, Technischen Universität München (1999)
11. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning 44(4), 303–336 (2010)
12. 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)

13. Hurd, J.: OpenTheory: Package management for higher order logic theories. In: Reis, G.D., Théry, L. (eds.) PLMMS, pp. 31–37. ACM (August 2009)
14. Dennis, L.A., Collins, G., Norrish, M., Boulton, R.J., Slind, K., Robinson, G., Gordon, M., Melham, T.: The PROSPER Toolkit. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 78–92. Springer, Heidelberg (2000)
15. Sutcliffe, G.: The TPTP World – Infrastructure for Automated Reasoning. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 1–12. Springer, Heidelberg (2010)
16. Rushby, J.: An Evidential Tool Bus. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, p. 36. Springer, Heidelberg (2005)
17. Zimmer, J., Autexier, S.: The MathServe System for Semantic Web Reasoning Services. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 140–144. Springer, Heidelberg (2006)
18. Mohamed, O.A., Muñoz, C., Tahar, S. (eds.): TPHOLs 2008. LNCS, vol. 5170. Springer, Heidelberg (2008)
19. van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.): ITP 2011. LNCS, vol. 6898. Springer, Heidelberg (2011)

Functional Programs: Conversions between Deep and Shallow Embeddings

Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. This paper presents a method which simplifies verification of deeply embedded functional programs. We present a technique by which proof-certified equations describing the effect of functional programs (shallow embeddings) can be automatically extracted from their operational semantics. Our method can be used in reverse, i.e. from shallow to deep embeddings, and thus for implementing certifying code synthesis: we have implemented a tool which maps HOL functions to equivalent Lisp functions, for which we have a verified Lisp runtime. A key benefit, in both directions, is that the verifier does not need to understand the operational semantics that gives meanings to the deep embeddings.

1 Introduction

For purposes of program verification, programs can be represented in theorem provers either in terms of syntax (a deep embedding), e.g. using an abstract datatype or as a string of ASCII characters

```
(defun APPEND (x y)
  (if (consp x)
      (cons (car x) (APPEND (cdr x) y))
      y))
```

or alternatively, directly as functions in the logic of a theorem prover (shallow embeddings),

$$\begin{array}{rcl}\mathsf{append}\ x\ y & = & \mathsf{if\ consp}\ x \neq \mathsf{nil\ then} \\ & & \quad \mathsf{cons}\ (\mathsf{car}\ x)\ (\mathsf{append}\ (\mathsf{cdr}\ x)\ y) \\ & & \mathsf{else}\ y\end{array}$$

Shallow embeddings are easier to work with. Consider e.g. proving associativity of `APPEND`. Proving this over the shallow embedding is straightforward.

$$\mathsf{append}\ x\ (\mathsf{append}\ y\ z) \ = \ \mathsf{append}\ (\mathsf{append}\ x\ y)\ z$$

Proving the same for a deep embedding, w.r.t. an operational semantics $\xrightarrow{\mathsf{ev}}$, involves a tedious proof over a transition system: for all *res*, *env*, x, y, z,

$$\begin{array}{l}(\mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [x, \mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [y,z]], env, s) \xrightarrow{\mathsf{ev}} (res, s) \iff \\ (\mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [\mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [x,y], z], env, s) \xrightarrow{\mathsf{ev}} (res, s)\end{array}$$

In some cases, proofs over deep embeddings are unavoidable, e.g. if we are to connect the verification proof to the correctness theorem of a verified compiler or runtime, since these are stated in terms of semantics of deep embeddings.

This paper presents a novel proof-producing technique for converting between the two forms of embedding. Our conversions produce a proof for each run; the result is a *certificate theorem* relating the shallow embedding (append) to the deep embedding ($\texttt{APPEND}$) w.r.t. an operational semantics $\xrightarrow{\mathsf{ap}}$ which specifies how deeply embedded programs evaluate. This will be explained in Section 2.

$$\begin{array}{l} \forall x\ y\ \mathit{state}. \\ \quad \mathsf{code\_for\_append\_in}\ \mathit{state} \implies \\ \quad (\mathsf{Fun}\ \texttt{"APPEND"}, [x,y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \mathit{state}) \end{array}$$

The proof-producing translation technique described in this paper means that the verifier can let automation deal with the operational semantics and thus avoid even understanding the definition of the operational semantics. This work has applications in verification and code synthesis.

Program verification: Given a functional programs written in a deep embedding, e.g. ASCII, we can parse this into an abstract syntax tree and use the translation from deep to shallow to simplify verification. We have used this technique to significantly simplify the task of verifying a 2,000-line Lisp program, the Milawa theorem prover [5], w.r.t a semantics of Lisp [6].

Program synthesis. The ability to translate shallow embeddings into certified deep embeddings can be used to implement high-assurance code synthesis. We have implemented such proof-producing code synthesis from HOL4 into our previously verified Lisp runtime [6]. This improves on the trustworthiness of current *program extraction* mechanisms in HOL4, Isabelle/HOL and Coq which merely print functions into the syntax of SML, Ocaml, Haskell or Lisp without any assurance proof or connection to a semantics of the target language.

2 From Deep to Shallow Embeddings

For purposes of brevity and clarity we will base our examples in this paper on the following abstract syntax for Lisp programs. This is a subset of the input language of our verified Lisp runtime [6].

$$\begin{array}{lcl} \mathit{term} & ::= & \mathsf{Const}\ \mathit{sexp} \mid \mathsf{Var}\ \mathit{string} \\ & \mid & \mathsf{If}\ \mathit{term}\ \mathit{term}\ \mathit{term} \mid \mathsf{App}\ \mathit{func}\ (\mathit{term}\ \mathsf{list}) \mid \ldots \\ \mathit{func} & ::= & \mathsf{PrimitiveFun}\ \mathit{prim} \mid \mathsf{Fun}\ \mathit{string} \mid \mathsf{Funcall} \mid \mathsf{Define} \mid \ldots \\ \mathit{prim} & ::= & \mathsf{Cons} \mid \mathsf{Car} \mid \mathsf{Cdr} \mid \mathsf{Add} \mid \ldots \\ \mathit{sexp} & ::= & \mathsf{Val}\ \mathit{nat} \mid \mathsf{Sym}\ \mathit{string} \mid \mathsf{Dot}\ \mathit{sexp}\ \mathit{sexp} \end{array}$$

We define the operational semantics for this language using inductively defined relations: apply $\xrightarrow{\mathsf{ap}}$ and eval $\xrightarrow{\mathsf{ev}}$. Term *exp* evaluates, in environment *env*, to x (of type *sexp*) if $(\mathit{exp}, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (x, \mathit{new\_state})$; and the application of function f to arguments xs evaluates to x if $(f, xs, \mathit{state}) \xrightarrow{\mathsf{ap}} (x, \mathit{new\_state})$. Certain functions, e.g. Define, Print and Error, alter *state*.

2.1 Method

When converting deep embeddings into shallow embeddings our task is to derive a definition of a function append and prove a connection between them, e.g.

$$(\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \mathit{state})$$

The method by which we accomplish this has two phases. The first phase derives a theorem of the following form, for some *hypothesis* and *expression*.

$$\mathit{hypothesis} \implies (\mathit{body}, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{expression}, \mathit{state})$$

This derivation proceeds as a bottom-up traversal of the abstract syntax tree for the *body* of the function we are extracting. At each stage a lemma is applied to introduce the relevant syntax in *body* and, at the same time, construct the corresponding shallowly embedded operations in *expression*.

The second phase defines a shallow embedding using *expression* as the right-hand side of the definition and discharges (most of) the *hypothesis* using the induction that arises from the termination proof for the shallow embedding.

There is no guess work or heuristics involved in this algorithm, which means that well-written implementations can be robust.

2.2 Example: Append Function

An example will illustrate this algorithm. Consider APPEND from above. For the *first phase*, we aim to derive a theorem describing the effect of evaluating the body of the APPEND function, i.e.

$$\begin{array}{l} \mathsf{If}\ (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Consp})\ [\mathsf{Var}\ \texttt{"X"}]) \\ \quad (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Cons})\ [\ldots, \mathsf{App}\ (\mathsf{Fun}\ \texttt{"APPEND"})\ [\ldots]]) \\ \quad (\mathsf{Var}\ \texttt{"Y"}) \end{array} \tag{1}$$

Our bottom-up traversal starts at the leaves. Here we have variable look-ups and thus instantiate v to "X" and "Y" in the following lemma to get theorems describing the leaves of the program.

$$v \in \mathsf{domain}\ \mathit{env} \implies (\mathsf{Var}\ v, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{env}\ v, \mathit{state})$$

Now that we have theorems describing the leaves, we can move upwards and instantiate lemmas for primitives, e.g. for Cdr using modus ponens against:

$$\begin{array}{l} (\mathit{hyp} \implies (x, \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathit{exp}, \mathit{state})) \implies \\ (\mathit{hyp} \implies (\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Cdr})\ [x], \mathit{env}, \mathit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{cdr}\ \mathit{exp}, \mathit{state})) \end{array}$$

When we encounter the recursive call to APPEND we, of course, do not have a description yet. In this case, we insert a theorem where *hypothesis* makes the assumption that some function variable *append* describes this application.

$$\begin{array}{l} (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathit{append}\ x\ y, \mathit{state}) \implies \\ (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathit{append}\ x\ y, \mathit{state}) \end{array}$$

The result of the *first phase* is a theorem of the form

$$\textit{hypothesis} \implies (\textit{body}, \textit{env}, \textit{state}) \xrightarrow{\mathsf{ev}} (\textit{expression}, \textit{state})$$

Here *body* is the abstract syntax tree for the body of `APPEND`; and *expression* is the following, if $\textit{env} = \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}$,

$$\begin{array}{l} \mathsf{if\ consp}\ x \neq \mathsf{nil\ then} \\ \quad \mathsf{cons}\ (\mathsf{car}\ x)\ (\textit{append}\ (\mathsf{cdr}\ x)\ y) \\ \mathsf{else}\ y \end{array} \tag{2}$$

and, with the same *env* instantiation, *hypothesis* is:

$$\begin{array}{l} \mathsf{consp}\ x \neq \mathsf{nil} \implies \\ (\mathsf{Fun}\ \texttt{"APPEND"}, [\mathsf{cdr}\ x, y], \textit{state}) \xrightarrow{\mathsf{ap}} (\textit{append}\ (\mathsf{cdr}\ x)\ y, \textit{state}) \end{array}$$

Next we enter the *second phase*: we define append so that its right-hand side is (2) with *append* replaced by append. As part of the straightforward termination proof for this definition, we get an induction principle

$$\begin{array}{l} \forall P. \\ \quad (\forall x\ y.\ (\mathsf{consp}\ x \neq \mathsf{nil} \implies P\ (\mathsf{cdr}\ x)\ y) \implies P\ x\ y) \implies \\ \quad (\forall x\ y.\ P\ x\ y) \end{array} \tag{3}$$

which we will use to finalise the proof of the certificate theorem as follows.

For the running example, let P abbreviate the following.

$$\lambda x\ y.\ (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \textit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \textit{state})$$

We now restate the result of phase one using P and the definition of append:

$$\begin{array}{ll} \forall x\ y. & (\mathsf{consp}\ x \neq \mathsf{nil} \implies \mathsf{P}\ (\mathsf{cdr}\ x)\ y) \implies \\ & (\textit{body}, \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}, \textit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{append}\ x\ y, \textit{state}) \end{array} \tag{4}$$

Let code_for_append_in *state* state that the deep embedding (1) is bound to the name `APPEND` and parameter list [`"X"`, `"Y"`] in *state*. Now the operational semantics' rule for function application (Sec. 4.2 of [6]) gives us the following lemma.

$$\begin{array}{ll} \forall x\ y. & (\textit{body}, \{\texttt{"X"} \mapsto x, \texttt{"Y"} \mapsto y\}, \textit{state}) \xrightarrow{\mathsf{ev}} (\mathsf{append}\ x\ y, \textit{state}) \wedge \\ & \mathsf{code\_for\_append\_in}\ \textit{state} \implies \mathsf{P}\ x\ y \end{array} \tag{5}$$

By combining (4) and (5) we can prove:

$$\begin{array}{ll} \forall x\ y. & \mathsf{code\_for\_append\_in}\ \textit{state} \implies \\ & (\mathsf{consp}\ x \neq \mathsf{nil} \implies \mathsf{P}\ (\mathsf{cdr}\ x)\ y) \implies \mathsf{P}\ x\ y \end{array} \tag{6}$$

And a combination of (3) and (6) gives us:

$$\forall x\ y.\ \mathsf{code\_for\_append\_in}\ \textit{state} \implies \mathsf{P}\ x\ y \tag{7}$$

An expansion of the abbreviation P shows that (7) is the certificate theorem we were to derive for `APPEND`: it states that the shallow embedding append is an accurate description of the deep embedding `APPEND`.

$$\begin{array}{l} \forall x\ y\ \textit{state}. \\ \quad \mathsf{code\_for\_append\_in}\ \textit{state} \implies \\ \quad (\mathsf{Fun}\ \texttt{"APPEND"}, [x, y], \textit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{append}\ x\ y, \textit{state}) \end{array}$$

2.3 Example: Reverse Function

Now consider an implementation for `REVERSE` which calls `APPEND`. In the first phase of the translation, the certificate theorem for `APPEND` (from above) can be used to give a behaviour to Fun `"APPEND"`. The second phase follows the above proof very closely. The result is the following shallow embedding,

$$\begin{array}{lcl} \mathsf{reverse}\ x & = & \mathsf{if\ consp}\ x \neq \mathsf{nil\ then} \\ & & \quad \mathsf{append\ (reverse\ (cdr}\ x\mathsf{))\ (cons\ (car}\ x\mathsf{)\ nil)} \\ & & \mathsf{else\ nil} \end{array}$$

and a similar certificate theorem:

$$\begin{array}{l} \forall x\ \mathit{state}. \\ \quad \mathsf{code\_for\_reverse\_in}\ \mathit{state} \implies \\ \quad (\mathsf{Fun}\ \texttt{"REVERSE"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{reverse}\ x, \mathit{state}) \end{array}$$

Here code_for_reverse_in *state* also requires that code for `APPEND` is present.

2.4 More Advanced Language Features

The most advanced feature our Lisp language supports is dynamic function calls using Funcall: the name of the function to be called is the first argument to Funcall. The equivalent in ML is a call to a function variable. The difference is that Funcall is potentially unsafe, e.g. if called with an invalid function name or with the wrong number of arguments. (ML's type system prevents such unsafe behaviour in ML.) We can support Funcall as follows. First two definitions:

$$\begin{array}{r} \mathsf{funcall\_ok}\ \mathit{args}\ \mathit{state} = \exists v.\ (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (v, \mathit{state}) \\ \mathsf{funcall}\ \mathit{args}\ \mathit{state} = \varepsilon v.\ (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (v, \mathit{state}) \end{array}$$

We use the following lemma in the first phase of the translation algorithm whenever Funcall is encountered.

$$\mathsf{funcall\_ok}\ \mathit{args}\ \mathit{state} \implies (\mathsf{Funcall}, \mathit{args}, \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{funcall}\ \mathit{args}\ \mathit{state}, \mathit{state})$$

The result from phase two is a certificate theorem containing a side-condition which collects the hypothesis that the induction is unable to discharge, e.g. if we were translating a function `CALLF` that uses Funcall then we get:

$$\begin{array}{l} \forall x\ \mathit{state}. \\ \quad \mathsf{code\_for\_callf\_in}\ \mathit{state} \wedge \mathsf{callf\_side}\ x\ \mathit{state} \implies \\ \quad (\mathsf{Fun}\ \texttt{"CALLF"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{callf}\ x\ \mathit{state}, \mathit{state}) \end{array}$$

So far we have only considered pure functions, i.e. functions that don't alter *state*. Impure functions are also supported: they translate into shallow embeddings that take the *state* as input and produce a result pair: the return value and the new state, e.g. $(\mathsf{Fun}\ \texttt{"IMPURE\_FUN"}, [x], \mathit{state}) \xrightarrow{\mathsf{ap}} (\mathsf{impure\_fun}\ x\ \mathit{state})$.

3 From Shallow to Deep Embeddings

The description above explains how proof-producing translations from deep to shallow embeddings can be performed. The same algorithm can be used for translations in the opposite direction: start by inventing a deep embedding corresponding to the given shallow embedding and, at phase two, refrain from inventing a shallow embedding, instead use the given shallow embedding and its induction principle.

4 Summary and Related Work

This paper has presented a proof-producing algorithm for translating between shallow and deep embeddings of untyped first-order Lisp programs.

Trustworthy program synthesis is one application area of this work. Li et al. [4] have worked on compiling shallowly embedded functions into assembly code directly from HOL. In this paper we instead establish a connection between HOL and a high-level language (which has a verified runtime). Work by Hardin et al. [2] on decompiling Guardol programs has similar goals.

Program verification is another application area of this work. In this area, Charguéraud [1] has proposed a completely different way of verifying deep embeddings of functional programs. Charguéraud proposes that reasoning is to be carried out using *characteristic formulae* for functional programs. These formulas provide a way of unrolling the operational semantics without dealing with the operational semantics directly. His approach does not require functions to be total, unlike our approach. However, his technique provides relations, while our approach produces equations which fit better with powerful rewriting tactics.

The algorithm presented here bears some resemblance to work by Krauss et al. [3] on constructing termination proofs from termination of rewriting systems.

References

1. Charguéraud, A.: Program verification through characteristic formulae. In: International Conference on Functional Programming (ICFP). ACM (2010)
2. Hardin, D., Slind, K., Whalen, M., Pham, T.-H.: The Guardol Language and Verification System. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 18–32. Springer, Heidelberg (2012)
3. 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)
4. Li, G., Owens, S., Slind, K.: Structure of a Proof-Producing Compiler for a Subset of Higher Order Logic. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 205–219. Springer, Heidelberg (2007)
5. Myreen, M.O., Davis, J.: `http://www.cl.cam.ac.uk/~mom22/jitawa/`
6. Myreen, M.O., Davis, J.: A Verified Runtime for a Verified Theorem Prover. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 265–280. Springer, Heidelberg (2011)

Author Index