Marko Van Eekelen
Herman Geuvers
Julien Schmaltz
Freek Wiedijk (Eds.)

LNCS 6898

Interactive Theorem Proving

Second International Conference, ITP 2011
Berg en Dal, The Netherlands, August 2011
Proceedings

Springer

Lecture Notes in Computer Science 6898

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

Marko Van Eekelen   Herman Geuvers
Julien Schmaltz   Freek Wiedijk (Eds.)

Interactive Theorem Proving

Second International Conference, ITP 2011
Berg en Dal, The Netherlands, August 22-25, 2011
Proceedings

Springer

Volume Editors

Marko Van Eekelen
Open Universiteit, Faculteit Informatica
Postbus 2960, 6401 DL Heerlen, The Netherlands
E-mail: marko.vaneekelen@ou.nl

Herman Geuvers
Radboud Universiteit Nijmegen, FNWI/ICIS/IS
Postbus 9010, 6500 GL Nijmegen, The Netherlands
E-mail: herman@cs.ru.nl

Julien Schmaltz
Open Universiteit, Faculteit Informatica
Postbus 2960, 6401 DL Heerlen, The Netherlands
E-mail: julien.schmaltz@ou.nl

Freek Wiedijk
Radboud Universiteit Nijmegen, FNWI/ICIS/IS
Postbus 9010, 6500 GL Nijmegen, The Netherlands
E-mail: freek@cs.ru.nl

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-22862-9 e-ISBN 978-3-642-22863-6
DOI 10.1007/978-3-642-22863-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011933581

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

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

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 2011: the Second International Conference on Interactive Theorem Proving. It was held during August 22–25, 2011 in Berg en Dal, The Netherlands.

ITP brings together researchers working in all areas of interactive theorem proving. ITP is the evolution of the TPHOLs conference series to the broad field of interactive theorem proving. The inaugural meeting of ITP was held during July 11–14, 2010 in Edinburgh, Scotland, as part of the Federated Logic Conference (FLoC, July 9–21, 2010). TPHOLs meetings took place every year from 1988 until 2009.

There were 50 submissions to ITP 2011, each of which was reviewed by at least four Program Committee members. Out of the 50 submissions, 42 were regular papers and 8 were rough diamonds. The Program Committee accepted 21 regular papers, including one proof pearl and four rough diamonds. All 25 papers are included in the proceedings. The Program Committee also invited two leading researchers from Industry, Georges Gonthier (Microsoft Research) and Mike Kishinevsky (Intel Corporation), and two leading researchers from academia, Don Batory (University of Texas at Austing) and Bart Jacobs (Radboud University Nijmegen), to present invited lectures.

Two system demos were given at ITP 2011. Each demo consisted in an in-depth presentation of 90 minutes about the application of an ITP system on a real example. Details about the practical use of ACL2 and KeY were presented.

ITP 2011 featured seven associated workshops that took place on August 26 and August 27. The workshops were the following: The Third Coq Workshop, the Third Workshop on Dependently Typed Programming, the 10th KeY Symposium, the 6th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, the ITP 2011 Workshop on Mathematical Wikis, the Third Workshop on Modules and Libraries for Proof Assistants, the 6th International Workshop on Systems Software Verification.

We would like to thank our Local Chair Nicole Messink for the valuable and efficient support in planning and running ITP. We would like to thank all the local organizers for their help during the event.

The work of the Program Committe 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 ITP 2010 and TPHOLs and its predecessors since 1993.

Finally, we would like to thank our sponsors: The Netherlands Organisation for Scientific Research (NWO) and The Royal Netherlands Academy of Arts and Sciences (KNAW).

June 2011

Herman Geuvers
Freek Wiedijk
Marko Van Eekelen
Julien Schmaltz

Organization

Program Committee

David Aspinall	University of Edinburgh, UK
Jeremy Avigad	Carnegie Mellon University, USA
Stefan Berghofer	Technische Universität München, Germany
Yves Bertot	INRIA, France
Sandrine Blazy	IRISA - Université Rennes 1, France
Jens Brandt	University of Kaiserslautern, Germany
Jared Davis	The University of Texas at Austin, USA
Amy Felty	University of Ottawa, Canada
Jean-Christophe Filliâtre	CNRS, France
Herman Geuvers	Radboud University Nijmegen, The Netherlands
Elsa Gunther	University of Illinois at Urbana-Champaign, USA
John Harrison	Intel Corporation, USA
Reiner Hähnle	Chalmers University of Technology, Sweden
Matt Kaufmann	University of Texas at Austin, USA
Gerwin Klein	NICTA and UNSW, Australia
Assia Mahboubi	INRIA Saclay – Île-de-France, France
Panagiotis Manolios	Northeastern University, USA
John Matthews	Galois Connections, Inc., USA
Paul Miner	NASA, USA
J Moore	University of Texas at Austin, USA
Greg Morrisett	Harvard University, USA
Magnus O. Myreen	University of Cambridge, UK
Tobias Nipkow	Technische Universität München, Germany
Michael Norrish	NICTA, Australia
Sam Owre	SRI International, USA
Christine Paulin-Mohring	Université Paris-Sud, France
Lawrence Paulson	University of Cambridge, UK
Brigitte Pientka	McGill University, Canada
Lee Pike	Galois, Inc., USA
Sandip Ray	University of Texas at Austin, USA
Jose-Luis Ruiz-Reina	University of Seville, Spain
David Russinoff	AMD, USA
Julien Schmaltz	Open University of The Netherlands / Radboud University Nijmegen, The Netherlands
Konrad Slind	Rockwell Collins Advanced Technology Center

Sofiène Tahar	Concordia University, Canada
Marko Van Eekelen	Radboud University Nijmegen, The Netherlands
Makarius Wenzel	University of Paris Sud, France
Freek Wiedijk	Radboud University Nijmegen, The Netherlands

Additional Reviewers

Abbasi, Naeem
Ahrendt, Wolfgang
Akbarpour, Behzad
Andronick, June
Baelde, David
Bai, Yu
Bardou, Romain
Bobot, Francois
Boespflug, Mathieu
Bosma, Wieb
Campbell, Brian
Cave, Andrew
Chamarthi, Harsh Raju
Cohen, Cyril
Contejean, Evelyne
Delahaye, David
Diatchki, Iavor
Dixon, Lucas
Erkok, Levent
Forest, Julien
Goodloe, Alwyn
Gunter, Elsa
Hasan, Osman
Hendrix, Joe
Herencia-Zapana, Heber
Huffman, Brian
Hurd, Joe
Jain, Mitesh
Ji, Ran
Kersten, Rody
Khan-Afshar, Sanaz
Klebanov, Vladimir
Krauss, Alexander
Kunz, César
Lensink, Leonard
Liu, Hanbing
Longuet, Delphine
Madlener, Ken
Maier, Patrick
Martin-Mateos, Francisco-Jesus
McKinna, James
Mhamdi, Tarek
Munoz, Cesar
Naumann, David
Papavasileiou, Vasilis
Pasca, Ioana
Pollack, Randy
Popescu, Andrei
Pottier, Loïc
Preoteasa, Viorel
Rager, David
Rutten, Luc
Sacchini, Jorge Luis
Schaefer, Ina
Schfer, Jan
Seidel, Peter-Michael
Sewell, Thomas
Shankar
Smetsers, Sjaak
Spitters, Bas
Starostin, Artem
Tankink, Carst
Tews, Hendrik
Théry, Laurent
Urban, Christian
Verbeek, Freek
Weber, Tjark
Whiteside, Iain
Winwood, Simon
Wolff, Burkhart

Table of Contents

Invited Papers

Regular Papers

Proof Pearls

Rough Diamonds

Towards Verification of Product Lines*
(Abstract)

Don Batory

Department of Computer Science
The University of Texas at Austin
Austin, Texas, USA
batory@cs.utexas.edu

Abstract. Although mechanized proof assistants are powerful verification tools, proof development can still be difficult and time-consuming. It becomes even more challenging when proofs are needed for product lines. A *product line* is a family of similar programs. Each program is constructed by a distinct (linear) combination of features, where a *feature* or *feature module* encapsulates program fragments that are added to a program to introduce a new capability or functionality to that program.

The first part of my presentation reviews basic concepts on product lines and how programs are synthesized using feature modules. The second part addresses product line verification. I explain how proofs for product lines can be engineered using feature modules. Each module contains proof fragments which are composed to build a complete proof of correctness for each product. A product line of programming languages is considered, where each variant includes metatheory proofs verifying the correctness of its syntax and semantic definitions. The approach has been realized in the Coq proof assistant, where the proofs of each feature are independently certifiable by Coq. Proofs are composed for each language variant, where Coq mechanically verifies that the composite proofs are correct. As validation, a core calculus for Java in Coq was formalized which can be extended with any combination of casts, interfaces, or generics.

Acknowledgements. Delaware, Cook and Batory are supported by the NSF's Science of Design Project CCF 0724979.

Reference

1. Delaware, B., Cook, W.R., Batory, D.: Theorem Proving for Product Lines. Tech. Rep. TR-11-25, University of Texas at Austin, Dept. of CS (May 2011)

[*] This is joint work with Benjamin Delaware and William Cook [1].

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, p. 1, 2011.

Advances in the Formalization of the Odd Order Theorem

Georges Gonthier

Microsoft Research Cambridge
gonthier@microsoft.com

Abstract. We present some of the proof techniques and library designs we used to formalize a large part of the proof of the Odd Order theorem.

Keywords: Formalization of Mathematics, Group Theory, Algebra, proof library, Coq, ssreflect.

The Odd Order theorem states that all finite groups of odd order are solvable. Due to Feit and Thompson, this very important and useful result in Group Theory is also historically significant because it initiated the large collective effort that lead to the full classification of finite simple groups twenty years later. It is also one of the first proofs to be questioned by prominent mathematicians because of its sheer length (255 pages) and complexity. These qualities make the Odd Order theorem a prime example for demonstrating the applicability of computer theorem proving to graduate and research-level mathematics.

As the Feit-Thompson proof draws on an extensive set of results spanning most of undergraduate algebra and graduate finite group theory, we would have to develop a substantial library of mathematical results to cover the prerequisites. We hoped that this library, its architecture, and the techniques supporting it, would provide practical outputs. Our starting point was the combinatorics library of the four-color theorem proof, and the small-scale reflection technique and the structured proof scripting language it used — these became the ssreflect extension to Coq.

Small-scale reflection consists in using the algorithmic fragment of the proof system logic (i.e., CiC for Coq) to capture more of the operational content of a mathematical theory (its "exercises"). Although reflection could not be used directly as often for general algebra than for combinatorics, we found that in association with extended type inference (supporting type classes) reflection could be used to create generic or "overloaded" theorems and theories. We used this everywhere, to create a variety of useful components, for, e.g., summations, algebraic structures, linear spaces, groups, group morphisms, characteristic subgroups...

This groundwork was tested in 2010, as we formalized more than half of the proof (the local analysis part). The smoothness of the process validated the library design: we could formalize nearly two pages a day when we had the right prerequisites. The proof language handled gracefully two unexpected difficulties in the graduate material: large lemmas (proving over 20 assertions under more than four separate assumptions), and non-structural proof steps (casual use of induction, abduction and symmetry), neither of which appeared in more basic material.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, p. 2, 2011.

Logical Formalisation and Analysis of the Mifare Classic Card in PVS

Bart Jacobs and Ronny Wichers Schreur

Institute for Computing and Information Sciences, Radboud University Nijmegen
Heijendaalseweg 135, 6525AJ Nijmegen, The Netherlands
{bart,ronny}@cs.ru.nl

Abstract. The way that Mifare Classic smart cards work has been uncovered recently [6,8] and several vulnerabilities and exploits have emerged. This paper gives a precise logical formalisation of the essentials of the Mifare Classic card, in the language of a theorem prover (PVS). The formalisation covers the LFSR, the filter function and (parts of) the authentication protocol, thus serving as precise documentation of the card's ingredients and their properties. Additionally, the mathematics is described that makes two key-retrieval attacks from [6] work.

1 Introduction

Computer security is hard. Any small oversight during the design and implementation of a system can render it insecure. A determined attacker can, and will, use any weakness in the system to get access. Formal methods can be a great help for designers and implementers to obtain precise descriptions of systems and rigorous proofs of their security properties.

The benefits of formal methods already become apparent in making a precise mathematical description of the system. Experience shows that many errors are found during this description phase, even before any verification is attempted.

In the area of protocol analysis, formal methods have become an important tool. Here the proofs are normally *symbolic*: the cryptographic primitives such as encryption and decryption functions are assumed to be sound and the analysis focuses on security properties of the protocol itself. The tools for symbolic protocol analysis are typically automated. Examples include ProVerif [1], based on the pi calculus, and other tools based on model checking (see [5] for a survey). Paulson [10] is a prominent example of interactive symbolic verification, using Isabelle.

Apart from the symbolic approach, assuming perfect cryptography, there is the computational one, which is more realistic but makes verification more difficult. In this approach cryptographic primitives act on strings of bits and security is defined in terms of low probability of success for an attacker. Both protocols and attackers are modelled as probabilistic polynomial-time Turing machines.

More recently cryptographic schemes are verified via a formalisation of game-based probabilistic programs with an associated logic, so that standard patterns

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 3–17, 2011.

in provable security can be captured. This approach is well-developed in the CertiCrypt tool, integrated with the theorem prover Coq, see *e.g.* [3,2] (or [16] for an overview).

The security of a software system also depends on the correctness of its implementation. This analysis is part of the well-established field of *program correctness*. Many programming errors do not only limit the functionality of the system, but also lead to exploits. So, all the advances that have been made in the area of program verification are also relevant for security. Here both automated and interactive proof tools are employed.

It is important to realise the limitations of formal methods in the area of computer security. By nature, a formal system describes an abstraction of the actual system and the environment in which it operates. For example, there may be a formal proof of the correctness of a program, but an attacker can try to randomly flip bits by shooting a laser at the chip that runs the program. This will result in behaviour that the formal model does not capture, but that may allow the attacker to break the system.

Another disadvantage of applying formal methods are the costs. It is labour intensive and thus expensive to formalise a system and to prove its correctness. This is why formal verification is only required at the highest Evaluation Assurance Level (EAL7) in the Common Criteria certification[1].

The current paper gives another twist to the use of formal methods in computer security. Rather than proving a system secure, we describe and analyse attacks on an insecure system. To describe these attacks we give a formalisation of the system under attack. The formalisation of the attacks gives a direct connection between the attacks and properties of the system that enables them. In doing so, it clearly demonstrates the system's design failures. The formalisation also gives precise boundaries for the conditions under which the attacks apply.

The flawed system that we consider is the Mifare Classic. This is a contactless (RFID) smart card, sold by NXP (formerly Philips Semiconductors), that is heavily used in access control and public transport (like in London's Oyster card, Boston's Charlie card, and the Dutch OV-chipkaart). It is estimated that over 1 billion copies have been sold worldwide. The design goes back to the early nineties, predating Common Criteria evaluation practices in the smart-card area. The card (and Mifare readers) contains a proprietary encryption algorithm, called Crypto1. It uses a 48-bit linear feedback shift register (LFSR), a device that is well-studied in the literature (see *e.g.*, [13,15]), together with a special filter function that produces the keystream bits.

The security of the card relies partly on the secrecy of this algorithm. Details of Crypto1 have emerged, first after hardware analysis [8][2], and a bit later after a cryptanalysis [6]. The latter reference presents a mathematical model of the card, together with several attack scenarios for key retrieval. The current paper builds on [6] and elaborates certain mathematical (logical) details of this model and these attacks. It does not add new (cryptographic) results, but provides

[1] See `commoncriteriaportal.org`

[2] Made public in a presentation at the Chaos Computer Club, Berlin, 27/12/07.

further clarity about the card. In doing so it points out where design flaws reside and how they can be exploited.

In general, theorem provers provide machine support for the formalisation and verification of systems and their properties. They are used both for hardware and for software. A theorem prover may be seen as a sceptical colleague that checks and documents all individual proof steps and helps with tedious details. There are several sophisticated interactive theorem provers around, such as Isabelle [7,12], Coq [11], NQTHM [4] and PVS [9,14]. In this paper we (happen to) use PVS. But we do not rely on any special property or power of PVS. We shall try to abstract away from the specifics of PVS, and formulate results in the language of (dependently) typed higher-order logic, using a certain level of pretty printing. The point we wish to make is that using a theorem prover is useful (also) in the area of computer security, for a precise description and analysis of one's system. As such it may be used as part of precise documentation, or even as part of a certification procedure. As will be illustrated, this works well for relatively unsophisticated systems, like smart cards with low-level operations. The PVS formalisation is available on the web[3].

The formalisation presented in this paper is specific to the Mifare Classic card and so it does not carry over to other systems. There is little or no uniformity in (proprietary) cryptographic systems, and hence there can be no uniformity in their formalisations. In a broader context this paper sets out to show that formalisations contribute to the documentation and analysis of low-level security protocols. The method as such does apply to other systems.

The paper is organised as follows. It first describes some general properties of LFSRs and filter functions, focusing on what is relevant here, and not developing much meta-theory. Subsequently, Section 3 gives the crucial ingredients that model the stream cipher Crypto1 of the Mifare Classic card, and Section 4 shows how these operations can be rolled back. Section 5 illustrates how the definitions and results from Section 3 establish (part of) the correctness of the mutual authentication protocol between a card and a reader. Finally, Section 6 elaborates the mathematical properties underlying two attacks from [6], namely the "two-table" and "odd-from-even" attacks.

2 Shift Registers, Generally

This section describes the formalisation of shift registers in PVS, together with some basic properties. The feedback function will at this stage be a parameter. A concrete version will be provided in Section 3.

The formalisation uses the PVS type `bvec[N]` of bit vectors of length `N:nat`. The natural number `N` is thus a parameter. The type may be instantiated concretely as `bvec[10]`, which yields the type of bit vectors of length 10. The type `bvec[N]` is defined as the type of functions from natural numbers below `N` to the type `bit` = `bool`, with TRUE and FALSE as (only) inhabitants. One writes

[3] `http://www.cs.ru.nl/~ronny/mifare-pvs`

fill[N](b):bvec[N] for the constant bit vector filled with b:bit at every position.

The logical description of LFSRs at this stage contains two parameters, namely their length LfsrSize:posnat (a positive natural number) and a feedback function feedback:[bvec[LfsrSize]→bit] that maps a bit vector of length LfsrSize to a bit. For convenience we abbreviate this type of bit vectors as 'state' in a PVS type definition:

```
state : TYPE = bvec[LfsrSize]
```

Then we can define the basic "left shift" function that captures the operation of an LFSR. It is called shift1in because it takes one bit and puts it into the LFSR on the right, while shifting the whole LFSR one position to the left.

```
shift1in : [state, bit → state] =
  λ(r:state, b:bit) : λ(i:below(LfsrSize)) :
    IF i < LfsrSize - 1
    THEN r(i+1)                % shift left
    ELSE b XOR feedback(r)     % put new value at i = LfsrSize - 1
    ENDIF
```

A picture of a concrete LFSR appears in Figure 1 in Section 3. Notice that the leftmost bit at position 0 is dropped, and that a new bit is inserted at the rightmost position LfsrSize-1. Via recursion an "N-ary" version is defined in a straightforward manner:

```
shiftNin : [state, N:nat, bv:bvec[N] → state] = ...
```

One can then prove basic properties, like:

```
shiftNin(r, N, bv)(i) = r(i+N)
shiftNin(shiftNin(r, N1, bv1), N2, bv2) = shiftNin(r, N1+N2, bv1 o bv2)
```

where i < LfsrSize-N and o is concatenation of bit vectors.

During initialisation a Mifare card and reader each feed certain (nonce) data into their LFSRs, see Section 5; afterwards they use their LFSR to produce a keystream by feeding it with 0s. This is captured by a special 'advance' function in PVS that has the number n:nat of inserted zeros as argument:

```
advance : [state, nat → state] =
  λ(r:state, n:nat) : shiftNin(r, n, fill[n](FALSE))
```

It forms an action with respect to the monoid of natural numbers since it satisfies:

```
advance(r, 0) = r      advance(r, n+m) = advance(advance(r,n), m)
```

2.1 Adding a Filter Function Parameter

We remain a bit longer within the generic setting of LFSRs. We now add another parameter, namely a function filfun:[state→bit] that produces an output bit for an arbitrary state. A basic (single) step in the Mifare initialisation phase of card and reader involves processing one input bit while producing one (encrypted) output bit that is sent to the other side. There it is processed in a dual way, as described by the following two functions.

```
shiftinsend1 : [[state, bit] → [state,bit]] =
  λ(r:state, b:bit) : (shift1in(r,b), b XOR filfun(r))
receiveshiftin1 : [[state, bit] → state] =
  λ(r:state, b:bit) : shift1in(r, b XOR filfun(r))
```

These two functions satisfy the following "correctness" result.

```
∀(r:state, b:bit) :
  LET (r1,b1) = shiftinsend1(r,b) % b1 is transmitted, encrypted
  IN  receiveshiftin1(r,b1) = r1
```

This basic result requires some explanation: assume the two sides (card and reader) are in the same state `r` before performing these operations. Assume:

- one side (actually the reader) performs `shiftinsend1` and shifts one bit `b` into its state (leading to successor state `r1`), while transferring the encrypted version `b1 = b XOR filfun(r)` of `b` to the other side;
- the other side (the card) performs `receiveshiftin1` and receives this encrypted bit `b1`, decrypts it via `b1 XOR filfun(r)` and shifts it into its own state (which we assume to be equal `r`).

Then: both sides are again in the same post state, namely `r1`. Hence by performing these operations card and reader transfer data and remain in sync. In this way they communicate like via one-time pads, except that the keystream has cycles.

Also N-ary versions of the functions `shiftinsend1` and `receiveshiftin1` are defined, with appropriate properties. They are used in the following two functions

```
load_and_send_reader_nonce : [[state, nonce] → [state, nonce]] =
   λ(r:state, plain:nonce) : shiftinsendN(r, NonceSize, plain)
receive_reader_nonce : [state, nonce → state] =
  λ(r:state, cipher:nonce) : receiveshiftinN(r, NonceSize, cipher)
```

which will be used in the explanation of the Mifare authentication protocol in Section 5.

Of course, many more definitions and properties may be introduced for such abstract LFSRs. We confine ourselves to what is needed in our logical theory of the Mifare Classic. It includes a function to generate keystream bits, in the following way.

```
stream : [state, n:nat → bit] =
  λ(r:state, n:nat) :  filfun(advance(r,n))
```

It is used in a similar function `cipher` that not only produces keystream, but also the resulting state. It is defined with a dependent product type in:

```
cipher : [state, n:nat → [state, bvec[n]]] =
  λ(r:state, n:nat) : ( advance(r,n), λ(i:below(n)) : stream(r,i) )
```

It is well behaved, in the sense that it satisfies:

```
cipher(r, n+m) =
  LET (r1,c1) = cipher(r,n), (r2,c2) = cipher(r1,m) IN (r2, c2 o c1)
```

3 The Mifare LFSR and Filter Function

The parameters (like `LfsrSize`, `feedback` and `filfun`) that were used in the previous section are now turned into the specific values that they have in the Mifare Classic.

The sizes are easy:

```
LfsrSize : nat = 48     NonceSize : nat = 32
```

The Mifare feedback function, described as a generating polynomial, like in [8,6], is

$$g(x) = x^{48} + x^{43} + x^{39} + x^{38} + x^{36} + x^{34} + x^{33} + x^{31} + x^{29} + x^{24} + x^{23} + x^{21} + x^{19} + x^{13} + x^{9} + x^{7} + x^{6} + x^{5} + 1.$$

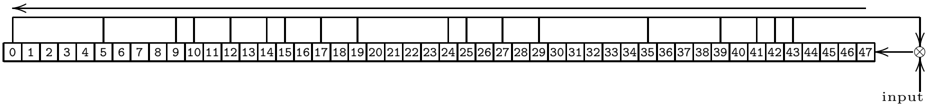


Fig. 1. Mifare Classic LFSR

In PVS this becomes, due to a reverse listing of entries:

```
MfCfeedback : [bvec[LfsrSize] → bit] =
  λ(r:bvec[LfsrSize]) :
    r(0) XOR r(5) XOR r(9) XOR r(10) XOR r(12) XOR r(14) XOR
    r(15) XOR r(17) XOR r(19) XOR r(24) XOR r(25) XOR r(27) XOR
    r(29) XOR r(35) XOR r(39) XOR r(41) XOR r(42) XOR r(43)
```

Notice that x^i corresponds to `r[48-i]`. The representation that we have chosen is the one that is most convenient in formulating definitions and properties. Now we can properly instantiate the theory of the previous section and obtain the type for “Mifare Classic LFSR” as:

```
MfClfsr : TYPE = state[LfsrSize, MfCfeedback]
```

We turn to the filter function for the Mifare Classic. It is constructed in several steps, via two auxiliary functions `MfCfilfunA` and `MfCfilfunB` that each produce one bit out of a 4-bit input. Such functions are usually described by 4 hexadecimal digits, capturing the conjunctive normal form. In this case we have `MfCfilfunA` = 0x26C7 and `MfCfilfunB` = 0x0DD3, which can be simplified to a disjunctive normal form:

```
MfCfilfunA(b3, b2, b1, b0:bit) : bit =
   ( (¬b3 ∧ ¬b2 ∧ ¬b1) ∨ (b3 ∧ ¬b1 ∧ b0) ∨ (¬b2 ∧ b1 ∧ ¬b0) ∨ (¬b3 ∧ b2 ∧ b1) )
MfCfilfunB(b3, b2, b1, b0:bit) : bit =
   ( (b3 ∧ ¬b2 ∧ ¬b0) ∨ (¬b3 ∧ b2 ∧ ¬b0) ∨ (b3 ∧ ¬b2 ∧ b1)
     ∨ (¬b3 ∧ b2 ∧ b1) ∨ (¬b3 ∧ ¬b2 ∧ ¬b1) )
```

The LFSR and filter function of the Mifare Classic can now be depicted in Figure 2.

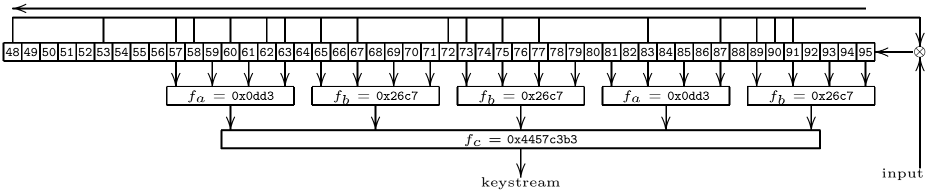


Fig. 2. Crypto1

One can then prove in PVS that these descriptions correspond to the conjunctive normal form given by the hexadecimal descriptions, but also to a "shift" description with the above values 0x26C7 and 0x0DD3: for a bit vector b of length 4,

```
MfCfilfunA(b(3), b(2), b(1), b(0))
  = right_shift(bv2nat(b), h2 o h6 o hC o h7)(0)
MfCfilfunB(b(3), b(2), b(1), b(0))
  = right_shift(bv2nat(b), h0 o hD o hD o h3)(0)
```

where bv2nat(b) gives the numerical value of the bit vector b, right_shift performs a number of shifts, as described by its first argument, and h2 etc. is the hexadecimal number 2, as bit vector of length 4 (with o describing concatenation, as before).

Two of these "A" and three "B" functions are combined into a new function that takes 20 bits input:

```
MfCfilfun20(b:bvec[20]) : bit =
  MfCfilfunC( MfCfilfunA(b(0), b(1), b(2), b(3)),
              MfCfilfunB(b(4), b(5), b(6), b(7)),
              MfCfilfunB(b(8), b(9), b(10), b(11)),
              MfCfilfunA(b(12), b(13), b(14), b(15)),
              MfCfilfunB(b(16), b(17), b(18), b(19)) )
```

where MfCfilfunC = 0x4457C3B3. Finally, the Mifare Classic filter function is described, following [6], as:

```
MfCfilfun(r:MfClfsr) : bit = MfCfilfun20(λ(i:below(20)) : r(9+2*i))
```

Important to note is the regularity of its application: on all odd positions 9, 11, 13, . . . , 47. This regularity is one of the weaknesses of the Mifare Classic, which can be exploited in various ways as we shall see in Section 6.

Finally, here are some test results that are proven in PVS simply by a single proof command "grind".

```
MfCfilfun(hA o hE o hA o h6 o h1 o hC o h9 o hC o h1 o hB o hF o h0) = 1
MfCfilfun(h7 o hD o hA o h8 o h8 o h0 o h1 o h8 o h8 o h6 o h1 o h5) = 0
```

4 Rollback Results

In this section we explore the structure described in the previous section, where we focus on the possibility of rolling back left shifts of the Mifare Classic register.

A first step is that we can recover the leftmost bit that is dropped in a single left shift step, if we know what the input bit is, via the following function.

```
leftmost : [MfClfsr, bit → bit] = λ(r:MfClfsr, b:bit) :
  r(47) XOR b XOR  % plus previous XORs, shifted one position
  r(4) XOR r(8) XOR r(9) XOR r(11) XOR r(13) XOR r(14) XOR
  r(16) XOR r(18) XOR r(23) XOR r(24) XOR r(26) XOR r(28) XOR
  r(34) XOR r(38) XOR r(40) XOR r(41) XOR r(42)
```

so that we can define an inverse of the `shift1in` function from Section 2 as:

```
shift1out : [MfClfsr, bit → MfClfsr] =
   λ(r:MfClfsr, b:bit) : λ(i:below(LfsrSize)) :
     IF i > 0
     THEN r(i-1)
     ELSE leftmost(r,b)      % at position i = 0
     ENDIF
```

and prove that they are indeed each other's inverses:

```
shift1out(shift1in(r, b), b) = r    shift1in(shift1out(r, b), b) = r
```

This shifting-out extends to an N-ary version, which is then inverse to N-ary shifting-in. Interestingly, the earlier advance function can now be extended from natural numbers to integers as:

```
Advance : [MfClfsr, int → MfClfsr] = λ(r:MfClfsr, n:int) :
    IF n ≥ 0
    THEN advance(r, n)
    ELSE shiftNout(r, -n, fill[-n](FALSE))
    ENDIF
```

We now get an action with respect to the monoid of integers:

```
Advance(r, 0) = r    Advance(r, i+j) = Advance(Advance(r,i), j)
```

where `i,j:int`. This allows us to smoothly compute a keystream not only in forward but also in backward direction.

4.1 Rolling Back Communication

So far we have concentrated on rolling back the LFSR. Since the Mifare Classic filter function MfCfilfun does not use the bit at position 0 it can also be reconstructed after a shift-left. This is another design error. We proceed as follows.

```
shiftout_MfCfilfun(r:MfClfsr) : bit =
  MfCfilfun20(λ(i:below(20)) : r(8+2*i))
shiftoutsend1 : [[MfClfsr, bit] → [MfClfsr,bit]] =
  λ(r1:MfClfsr, b1:bit) : LET b = b1 XOR shiftout_MfCfilfun(r1)
                          IN  (shift1out(r1,b), b)
```

Then we obtain an inverse to the basic step of the card from Subsection 2.1:

```
shiftoutsend1(shiftinsend1(r, b)) = (r, b)
shiftinsend1(shiftoutsend1(r, b)) = (r, b)
```

This can be done multiple times.

5 The Mifare Classic Authentication Protocol

When a card reader wants to access the information on a Mifare Card it must prove that it is allowed to do so. Conversely the card must prove that is a authentic card. Both security goals are accomplished by a mutual-authentication mechanism based on a symmetric-key cipher. Figure 3 pictures in detail how an authentication proceeds.

	Card		Reader
0		anti-c(uid) →	
1		auth(block) ←	
2	$S_1 \leftarrow K_{\mathsf{block}}$		$S_1 \leftarrow K_{\mathsf{block}}$
3	picks n_C		
4	$S_2 \leftarrow \mathsf{shiftinN}(S_1, \mathrm{uid} \oplus n_C)$		
5		n_C →	
6			$S_2 \leftarrow \mathsf{shiftinN}(S_1\ \mathrm{uid} \oplus n_C)$
7			picks n_R
8			$(S_3, \{n_R\}) \leftarrow \mathsf{send\_reader\_nonce}(S_2, n_R)$
9		$\{n_R\}$ ←	
10	$S_3 \leftarrow \mathsf{receive\_reader\_nonce}(S_2, \{n_R\})$		
11			$(S_4, \mathrm{ks}_2) \leftarrow \mathsf{cipher}(S_3)$
12		$\mathsf{suc}^2(n_C) \oplus \mathrm{ks}_2$ ←	
13	$(S_4, \mathrm{ks}_2) \leftarrow \mathsf{cipher}(S_3)$		
14	verify $\mathsf{suc}^2(n_C)$		
15	$(S_5, \mathrm{ks}_3) \leftarrow \mathsf{cipher}(S_4)$		
16		$\mathsf{suc}^3(n_C) \oplus \mathrm{ks}_3$ →	
17			$(S_5, \mathrm{ks}_3) \leftarrow \mathsf{cipher}(S_4)$
18			verify $\mathsf{suc}^3(n_C)$

Fig. 3. Mifare Classic Authentication Protocol

Because Mifare Classic cards operate through radio waves, it is possible that more than one card is within range of a reader. To distinguish different cards, each card has a unique id that is send to the reader (step 0). This 32-bit uid also plays a role in the cipher.

The information on the Mifare Classic is divided into blocks. The reader starts an authentication session for a specific memory block (step 1). Each block is protected by a different 48-bit key that is known by both the card and the reader. Card and reader initialise their shift registers with this key K_{block} (step 2).

The card subsequently chooses a 32-bit challenge or card nonce (n_C). This card nonce is added ($\oplus$) to the uid and the result is fed into the LFSR. Also the card nonce is send to the reader in the clear, which then also feeds $n_C \oplus \mathsf{uid}$ into its LFSR.

Then it is up to the reader to pick a 32-bit reader nonce n_R. This nonce is also fed into the LFSR. After each bit the output of the filter function is collected in the encrypted reader nonce $\{n_R\}$ (send_reader_nonce in step 8).

Upon reception of the encrypted reader nonce $\{n_R\}$ the card performs the inverse operation of send_reader_nonce, that is receive_reader_nonce.

At this moment (after step 10) the cipher is initialised. The keystream now consists of the output of the filter function after each shift of the LFSR. All further communication is encrypted by adding the keystream to the clear text. Decryption is simply adding the keystream to the cipher text.

The reader responds to the card's challenge by sending the encrypted card nonce n_C, or rather the encryption of the expression $\mathsf{suc}^2(n_C)$. The function suc is actually computed by another 16-bit LFSR that is used to generate the card nonces. The card can decrypt the reader's response and verify that it corresponds to the expected result. This establishes that the reader can correctly encrypt the challenge, which presumably means that the reader has knowledge of K_{block} and thus is allowed to access that block.

To complete the mutual authentication, the card returns the encryption of $\mathsf{suc}^3(n_C)$. The reader can verify that the card's response is properly encrypted, which implicitly established the authenticity of the card.

To show that a reader and a card can perform a successful mutual authentication, we can show that after each step they are both in the same state. In Figure 3 this means that every S_n on the card side is equal to the S_n on the reader side. For most steps this is easy, since the card and the reader perform identical operations. Only when the reader nonce is processed, do the card and reader operate differently. The following PVS theorem states the correctness property of this step.

```
∀(s2 : state, plain_reader_nonce : nonce) :
  LET
      (rs3, encrypted_reader_nonce)
         = load_and_send_reader_nonce(s2, plain_reader_nonce),
      cs3
         = receive_reader_nonce(s2, encrypted_reader_nonce)
  IN
      cs3 = rs3
```

6 Formalising Attacks

This section formalises the essentials of two attacks from [6]. They form a *post hoc* justification of the (C-code) implementation that underlies [6]. One can justifiably ask: what is the point of such a formalisation? After all, the attack in C can be executed and thus shows if it works or not. It does not need to work all the time, under all circumstances and only needs to work as a prototype[4], to show the feasibility of exploiting certain card vulnerabilities.

Our answer is that the formalisation explains the details—including assumptions and side-conditions—of the attacks and thus clearly demonstrates the precise vulnerabilities on which the attacks are built. This clarity may help to prevent or counter such vulnerabilities in similar situations.

6.1 The Two-Table Attack

The first attack that will be formalised comes from [6, §§6.3]. It exploits the fact that the filter function MfCfilfun acts on only twenty LFSR positions, which are all at regular, odd positions (9, 11, . . . , 47). Hence after shifting the LFSR two positions the filter functions gets very similar input.

This attack proceeds as follows. Assume we have a certain amount of keystream (at least 12 bits long). The aim is to find "solutions", namely LFSR states that produces this keystream, via the filter function. The first step is to define appropriate types for this setting:

```
keystream : TYPE =
  [# len : {n:posnat | even?(n) ∧ n ≥ 12}, bits : bvec[len] #]
solutions(ks : keystream) : PRED[MfClfsr] =
  { r : MfClfsr | ∀(i:below(len(ks))) : stream(r,i) = bits(ks)(i) }
```

The notation [# .. #] is used for labelled-product types. The length len(ks) of a keystream ks:keystream is thus even and bigger than 12, with a bit vector bits(ks) of this length.

For both the evenly and oddly numbered bits of keystream, we can look at the 20 bits of filter-function input that produce them. These will be described as even or odd "subsolutions", like in:

```
subsolutions_even(ks) : PRED[bvec[len(ks)/2 + 19]] =
  { s : bvec[len(ks)/2 + 19] | ∀(shiftnr:below(len(ks)/2)) :
       MfCfilfun20(λ(i:below(20)) : s(shiftnr + i))
         = bits(ks)(2*shiftnr) }
subsolutions_odd(ks) : PRED[bvec[len(ks)/2 + 19]] =
  { t : bvec[len(ks)/2 + 19] | ∀(shiftnr:below(len(ks)/2)) :
       MfCfilfun20(λ(i:below(20)) : t(shiftnr + i))
         = bits(ks)(2*shiftnr + 1) }
```

One sees that this formalisation makes the boundaries involved clearly visible.

[4] Unless one has malicious intentions.

In a next step we use the feedback function of the Mifare Classic LFSR to relate these two subsolutions. In order to do so we need to split the original feedback function, described in Section 3 as MfCfeedback, into two parts:

```
feedback_even(bv:bvec[24]) : bit =
  bv(0) XOR bv(5) XOR bv(6) XOR bv(7) XOR bv(12) XOR bv(21)
feedback_odd(bv:bvec[24]) : bit =
  bv(2) XOR bv(4) XOR bv(7) XOR bv(8) XOR bv(9) XOR bv(12) XOR
  bv(13) XOR bv(14) XOR bv(17) XOR bv(19) XOR bv(20) XOR bv(21)
```

in such a way that the original feedback is obtained as:

```
MfCfeedback(r) = (feedback_even(λ(i:below(24)) : r(2*i))
                 XOR feedback_odd(λ(i:below(24)) : r(2*i+1)))
```

The match that we seek between even and odd subsolutions is expressed by the following relation between two bit vectors.

```
shift2match?(ebv:bvec[25], obv:bvec[25]) : bool =
  feedback_even(λ(i:below(24)) : ebv(i))
      = (ebv(24) XOR feedback_odd(λ(i:below(24)) : obv(i)))
   ∧ feedback_even(λ(i:below(24)) : obv(i))
        = (obv(24) XOR feedback_odd(λ(i:below(24)) : ebv(i+1)))
```

It is used to define matching subsolutions for a given keystream:

```
subsolutions(ks) =
  { (s : (subsolutions_even(ks)), t : (subsolutions_odd(ks))) |
     ∀(shiftnr:below(len(ks)/2 - 5)) :
        shift2match?(λ(i:below(25)) : s(i+shiftnr),
                     λ(i:below(25)) : t(i+shiftnr)) }
```

The main result then says:

```
∀(st : (subsolutions(ks)), shiftnr:below(len(ks))) :
  MfCfilfun(Advance(merge(ks)(st), shiftnr-9)) = bits(ks)(shiftnr)
```

where the merge function yields an LFSR state:

```
merge(ks)(st) : MfClfsr = λ(i:below(LfsrSize)) :
  IF even?(i) THEN proj_1(st)(i/2) ELSE proj_2(st)(i/2 - 1) ENDIF
```

The main result expresses a correctness property: the bits of a keystream can be obtained by applying the filter function to a merge of matching (even and odd) subsolutions. As sketched in [6, §§6.3], the set of such subsolutions can be calculated efficiently, from which a merged LFSR state results. The above correctness result shows that this process can be seen as an inverse to the filter function of the Mifare Classic card.

6.2 The Odd-from-Even Attack

It is possible to improve upon the two-table attack when sufficiently many bits of the keystream are known. In the previous section we saw that subsolutions of even bits of the LFSR can be computed from the even bits of the keystream. Suppose we have an even subsolution of 48 bits. The following property specifies that a subsolution of 48 even bits corresponds to a given LFSR state.

```
StateAndEvens(r:MfClfsr, e:bvec[LfsrSize]) : bool
  = ∀(i:below(HalfLfsrSize)) :
      e(i) = r(2*i) ∧ e(HalfLfsrSize + i) = advance(r, LfsrSize)(2*i)
```

Given such a even subsolution, we can algebraically obtain the odd bits of the LFSR.

Consider as illustration the 4-bits LFSR with generating polynomial x^4+x^1+1. If the bits of `advance(r, LfsrSize)` are named $r_4..r_7$, we can express these in terms of initial LFSR bits $r_0..r_3$:

$$\begin{aligned}
r_4 &= r_0 \oplus r_3;\\
r_5 &= r_1 \oplus r_4 = r_0 \oplus r_1 \oplus r_3;\\
r_6 &= r_2 \oplus r_5 = r_0 \oplus r_1 \oplus r_2 \oplus r_3;\\
r_7 &= r_3 \oplus r_6 = r_0 \oplus r_1 \oplus r_2.
\end{aligned}$$

For an even subsolution the even variables r_0, r_2, r_4 and r_6 are known. This leaves us with a system of four linear equations in four unknowns (the odd variables r_1, r_3, $r5$ and r_7). Solving the system gives

$$\begin{aligned}
r_1 &= r_2 \oplus r_4 \oplus r_6;\\
r_3 &= r_0 \oplus r_4;\\
r_5 &= r_2 \oplus r_6;\\
r_7 &= r_0 \oplus r_4 \oplus r_6.
\end{aligned}$$

In particular, we now have expressed the missing odd bits of the initial LFSR ($r1$ and $r3$) in terms of the bits of the even subsolution.

This computation can also be performed for the Mifare Classic LFSR. For example the first odd bit equals $e_1 \oplus e_2 \oplus e_6 \oplus e_{11} \oplus e_{13} \oplus e_{19} \oplus e_{21} \oplus e_{22} \oplus e_{23} \oplus e_{25} \oplus e_{28} \oplus e_{30} \oplus e_{32} \oplus e_{34} \oplus e_{36} \oplus e_{37} \oplus e_{38} \oplus e_{39} \oplus e_{41} \oplus e_{42}$. The equations for the value of the odd bits in terms of the even bits were obtained with an external program, but their correctness can readily be verified within PVS.

```
∀(r:MfClfsr, e:bvec[LfsrSize]) :
  StateAndEvens(r,e) IMPLIES (
    (r(1) = (e(1) XOR e(2) XOR e(6) XOR e(11) XOR e(13) XOR
                e(19) XOR e(21) XOR e(22) XOR e(23) XOR e(25) XOR
                e(28) XOR e(30) XOR e(32) XOR e(34) XOR e(36) XOR
                e(37) XOR e(38) XOR e(39) XOR e(41) XOR e(42)))
     AND
% similarly for r(3), r(5) .. r(47)
```

These observations lead to the following efficient attack. Suppose we have 58 bits of keystream, that is we have 29 bits of even and odd keystream each. Using the 29 bits of even keystream we do a depth-first search to find the even subsolutions of 48 bits using the extension method of Section 6.1. For each even subsolution we can compute the odd bits of the LFSR. These in turn determine the odd bits of the keystream, which can be matched against the observed odd keystream.

This attack is more efficient, because the odd subsolution is obtained directly from the even subsolutions (an $O(1)$ operation) whereas in the two-table attack each even subsolution has to be matched against each odd subsolution ($O(n^2)$ when done naively, $O(n \log(n))$ using a sorting operation on the feedback values).

7 Conclusions

We have described the basic logical details of the Mifare Classic card, focussing on its vulnerabilities and on two exploits. In the theorem prover PVS we have proved essential correctness results, while abstracting away from for instance matters of efficiency.

Many of the details of the formalisation are inherently specific to the Mifare Classic card and its weaknesses. However, this work does show that formalisations are relatively easy to do and can be both precise and readable. This makes them a solid base for the documentation and analysis of cryptographic systems. Thus, this paper suggests to card producers that they do such formalisations themselves, before bringing a card onto the market.

Acknowledgments. The reverse engineering and cryptanalysis of the Mifare Classic chip was very much a team effort and we thank the other members of the Mifare team for their hard work. Peter van Rossum in particular is acknowledged for his contributions to this paper.

References

1. Abadi, M., Blanchet, B., Comon-Lundh, H.: Models and proofs of protocol security: A progress report. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 35–49. Springer, Heidelberg (2009)
2. Barthe, G., Daubignard, M., Kapron, B., Lakhnech, Y.: Computational indistinguishability logic. In: Computer and Communications Security, pp. 375–386. ACM, New York (2010)
3. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Principles of Programming Languages, pp. 90–101. ACM Press, New York (2009)
4. The Boyer-Moore theorem prover, `http://www.computationallogic.com/software/nqthm/`
5. Cortier, V., Kremer, S., Warinschi, B.: A survey of symbolic methods in computational analysis of cryptographic systems. Journ. Automated Reasoning 46(3-4), 225–259 (2010)
6. Garcia, F.D., de Koning Gans, G., Muijrers, R., van Rossum, P., Verdult, R., Schreur, R.W., Jacobs, B.: Dismantling MIFARE classic. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 97–114. Springer, Heidelberg (2008)

7. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. In: Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.) LNCS, vol. 2283, p. 3. Springer, Heidelberg (2002)
8. Nohl, K., Evans, D., Plötz, S., Plötz, H.: Reverse-engineering a cryptographic RFID tag. In: 17th USENIX Security Symposium, San Jose, CA, USA, pp. 185–194 (2008)
9. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
10. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journ. of Computer Security 6, 85–128 (1998)
11. The Coq proof assistant, `http://coq.inria.fr`
12. The Isabelle proof assistant, `http://isabelle.in.tum.de`
13. Solomon, W.G.: Shift register sequences. Aegean Park Press, Laguna Hills (1982)
14. The PVS Specification and Verification System, `http://pvs.csl.sri.com`
15. van Tilborg, H.C.A.: Fundamentals of Cryptology: a professional reference and interactive tutorial. Kluwer Academic Publishers, Dordrecht (2000)
16. Zanella Béguelin, S.: Formal Certification of Game-Based Cryptographic Proofs. PhD thesis, École Nationale Supérieure des Mines de Paris (2010)

Challenges in Verifying Communication Fabrics

Michael Kishinevsky[1], Alexander Gotmanov[2], and Yuriy Viktorov[2]

[1] Intel Corporation, Hillsboro, Oregon, USA
michael.kishinevsky@intel.com
[2] Intel Corporation, Moscow, Russia
{alexander.gotmanov,yuriy.viktorov}@intel.com

Abstract. Functional and performance correctness of on-die communication fabrics is critical for the design of modern computer systems. In this talk we will examine some challenges and open problems in functional verification communication fabrics and in their quality of service analysis and optimization. We will also review some progress that has been done in liveness verification of communication fabrics.

Keywords: liveness, deadlocks, communication fabrics, networks-on-chip, microarchitecture, high-level models, formal verification.

1 Communication Fabrics

Communication fabrics (a.k.a. interconnect fabrics or networks-on-chip) are critical for the quality (correctness, performance, energy, reliability) and fast integration of modern and future computer industry products. Examples of communication fabrics range from high-end regular rings and meshes in high-end servers, graphics and high-performance computing to SOC system agents handling coherent memory traffic, and to IO interconnect fabric.

Designing communication fabrics is one of the greatest challenges faced by designers of digital systems due a tricky distributed nature of communication and intricate nature of interaction between the micro-architecture of the fabric and higher-level protocols used by the fabric agents that need to be mapped over the fabric in correct and efficient manner. This challenge holds regardless of whether fabrics are regular or irregular in structure.

Due to the need to be compatible with existing protocols and design practices, early definition and exploration of interconnect fabrics are often based on previous designs. The detailed area and floorplan estimates done at the design phase require significant effort making these detailed estimates impractical at the architectural exploration stage where a designer needs to quickly evaluate and compare many architectures.

Existing verification techniques are insufficient to meet the validation challenges of demonstrating correctness of modern fabrics, such as deadlocks and livelocks in presence of message dependencies. Quality of service analysis, which is commonly done based on design review and performance modeling, can miss important corner cases.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 18–21, 2011.

Designing communication fabrics is a multidimensional challenge that involves complex functional and performance validation, cost analysis (area, power, design cost), and multilayered optimization (logical performance of interconnect vs. physical design aspects of the chip).

In this talk we will focus on challenges of formal verification of communication fabrics, i.e., proving their functional and performance correctness.

2 Verification Challenges

Time-to-market is one of the major constraints of system-on-chip (SoC) designs. Project teams usually rely on standard IPs re-used across multiple products to achieve quick turnaround time. Therefore, efficient system-level interconnect solutions that can seamlessly glue different IPs together become an integral part of designs. This design methodology trickles up to high-end server and high-performance computing systems where demand in reuse of standardized components (especially, the interconnect solutions) across multiple product segments and design generations is growing.

If not designed carefully, deadlocks and livelocks in system-level interconnect may present significant challenges to quick SoC integration. The distributed nature of the system-level interconnect and the complex interaction between many IPs make this problem hard to understand and debug.

Furthermore, deadlock problems at the system level are hard to fix or fix optimally after they are found. For example, optimal solutions may require altering many blocks, while simple solutions often sacrifice performance by limiting concurrency.

Recently, there has been significant progress in formal deadlock verification based on modeling systems using a well-defined set of functional primitives [6] and generating inductive invariants [5] and deadlock conditions statically capturing all possible system deadlocks [7]. A different promising technique uses embedding of generic verification conditions into a theorem prover [9].

However, there is a need to significantly extend verification methods to cover broader classes of design patterns, other liveness and safety properties such as cache coherency, producer-consumer relation, and memory consistency proofs taking into account micro-architectural details (unlike current state-of-the-art techniques for cache coherency verification).

In addition, new methods are needed for connecting high-level models to RTL either through optimizing fabric compilers or through generation of RTL validation observers.

3 xMAS Approach to Modeling and Verification

Our microarchitectural models are described by instantiating and connecting components from a library of primitives. We refer to these models as xMAS networks (xMAS stands for eXecutable MicroArchitectural Specification). The properties to be verified are specified on these networks. The semantics of xMAS networks are specified using synchronous equations for each primitive. Thus every xMAS network has an associated synchronous system which we call the *synchronous model*.[1] For verification, an xMAS network is compiled down into a synchronous model (single clock, edge-triggered Verilog to be precise) which is then verified.

We use the high-level structure of the xMAS primitives and models to discover system invariants which are then included into the synchronous model to ease the verification. The modeling methodology is described in more detail in [6] and its use in safety verification is described in [5].

In this talk we will review a lightweight, *automatic* approach that allows us to prove liveness on a large class of real examples drawn from the domain of communication fabrics. The definition of deadlock in xMAS models is *local*, i.e. it permits part of the model to be forever blocked while the rest continues processing packets. Such local deadlocks are also called livelocks.

The main idea behind our method is to exploit the high-level structure of the model in order to reason about liveness. We show that all non-live behaviors of xMAS network, or *structural deadlocks*, can be characterized by pure structural reasoning. Unreachable structural deadlocks are ruled out using safety invariants which are also obtained through automatic analysis of the model [7].

4 Quality of Service Analysis and Optimization

Current design practices in Quality of Service (QoS) analysis are predominantly restricted to two forms: informal design reviews and detailed performance models. The former is highly inaccurate and error prone, and the latter happens too late in the design since many of the important architectural decisions are locked-in by the time the performance model is completed. Moreover, neither of these methods can answer the system-level starvation problem nor compute the upper bound latencies for latency critical classes of traffic.

While similar problems have been in the past considered in network calculus [1], application of abstract interpretation to timing analysis of embedded systems [10], and analysis of event separation in asynchronous specifications and interfaces [4], neither of the above techniques appear to be directly applicable to modern on-die interconnects or they produce too loose bounds on performance guarantees.

Modern interconnect fabrics are parameterized (sizes of the queues, arbitration functions, channel widths, number of virtual classes). Quick selection of the parameters optimizing quality of service metrics is an open problem addressed in practice by design reviews and performance validation. Buffer optimization have been successfully considered for asynchronous [8] and synchronous elastic [2,3] specifications, however this only covers a small sub-class of design structures used in modern fabrics.

[1] If there is a combinational cycle in the synchronous model, the corresponding xMAS network is *ill-formed*.

Acknowledgment. We thank Satrajit Chatterjee for the development of xMAS system and the underlying theory, Umit Ogras for his contrubutions and Sayak Ray for implementing the liveness to safety conversion in ABC .

References

1. Boudec, J.Y.L., Thiran, P.: Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. LNCS, vol. 2050. Springer, Heidelberg (2001)
2. Bufistov, D., Júlvez, J., Cortadella, J.: Performance optimization of elastic systems using buffer resizing and buffer insertion. In: Proc. International Conf. Computer-Aided Design (ICCAD), pp. 442–448 (November 2008)
3. Bufistov, D.E., Cortadella, J., Galceran-Oms, M., Júlvez, J., Kishinevsky, M.: Retiming and recycling for elastic systems with early evaluation. In: DAC 2009: Proceedings of the 46th Annual Design Automation Conference, pp. 288–291. ACM, New York (2009)
4. Burns, S.M., Hulgaard, H., Amon, T., Borriello, G.: An algorithm for exact bounds on the time separation of events in concurrent systems. IEEE Trans. Comput. 44, 1306–1317 (1995), http://dx.doi.org/10.1109/12.475126
5. Chatterjee, S., Kishinevsky, M.: Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 321–338. Springer, Heidelberg (2010)
6. Chatterjee, S., Kishinevsky, M., Ogras, U.Y.: Quick formal modeling of communication fabrics to enable verification. In: Proc. IEEE High Level Design Validation and Test Workshop (HLDVT), pp. 42–49 (2010)
7. Gotmanov, A., Chatterjee, S., Kishinevsky, M.: Verifying deadlock-freedom of communication fabrics. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 214–231. Springer, Heidelberg (2011)
8. Manohar, R., Martin, A.J.: Slack elasticity in concurrent computing. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 272–285. Springer, Heidelberg (1998)
9. Verbeek, F., Schmaltz, J.: Formal specification of networks-on-chips: deadlock and evacuation. In: DATE 2010, pp. 1701–1706 (2010)
10. Wilhelm, R.: Timing analysis and timing predictability. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 317–323. Springer, Heidelberg (2005), http://dx.doi.org/10.1007/11561163_14

Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq

Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal

IT University of Copenhagen

Abstract. We present a shallow Coq embedding of a higher-order separation logic with nested triples for an object-oriented programming language. Moreover, we develop novel specification and proof patterns for reasoning in higher-order separation logic with nested triples about programs that use interfaces and interface inheritance. In particular, we show how to use the higher-order features of the Coq formalisation to specify and reason modularly about programs that (1) depend on some unknown code satisfying a specification or that (2) return objects conforming to a certain specification. All of our results have been formally verified in the interactive theorem prover Coq.

1 Introduction

Separation Logic [12,16] is a Hoare-style program logic for modular reasoning about programs that use shared mutable data structures. *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 post-conditions) and the specification logic (the logic of Hoare triples). HOSL was proposed with the purposes of (1) reasoning about data abstraction via quantification over resource invariants, and (2) making formalisations of separation logic easier by having one general expressive logic in which it is possible to define predicates, etc., needed for applications. In this article we explore these two purposes further; we discuss each in turn.

The first purpose (data abstraction) has been explored for a first-order language [4], for higher-order languages [9,11], and for reasoning about generics and delegates in object-oriented languages (without interfaces and without inheritance) [18]. In this article we show how HOSL can be used for modular reasoning about interfaces and interface-based inheritance in an object-oriented language like Java or $C\sharp$. Our current work is part of a research project in which we aim to formally specify and verify the C5 generic collection library [8], which is an extensive collection library that is used widely in practice and whose implementation makes extensive use of shared mutable data structures. A first case-study of one of the C5 data structures is described in [7]. C5 is written in $C\sharp$ and is designed mainly using interface inheritance, rather than class-to-class inheritance; different collection modules are related via an inheritance hierarchy among interfaces. For this reason we focus on verifying object-oriented programs that use interfaces and interface-based inheritance.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 22–38, 2011.

We explore the second purpose (formalisation) by developing a Coq formalisation of HOSL for an object-oriented class-based language and show through verified examples how it can be used to reason about interfaces and inheritance.

Our formalisation makes use of ideas from abstract separation logic [6] and thus consists of a general treatment of the assertion logic that works for many models and for a general operationally-inspired notion of semantic command. Our general treatment of the logic is also rich enough to cover so-called nested triples [17], which are useful for reasoning about unknown code, either in the form of closures or delegates [18] or, as we show here, in the form of code matching an interface. To reason about object-oriented programs, we instantiate the general development with the heap model for our object-oriented language and derive suitable proof rules for the language. This approach makes it easier in the future to experiment with other storage models and languages, e.g., variants of separation logic with fractional permissions.

Summary of contributions. We formalize a shallow Coq embedding of a higher-order separation logic for an object-oriented programming language. We have designed a system that allows us to write programs together with their specifications, and then prove that each program conforms to its specification. All meta-theoretical results have been verified in Coq[1].

We introduce a pattern for interface specifications that allows for a modular design. An interface specification is parametrised in such a way that any class implementing the interface can be given a suitably expressive specification by a simple instantiation of the interface specification. Moreover, we show how to use nested triples to, e.g., write postconditions in the assertion logic that require a returned object to match a certain specification. Our approach enables us to verify dynamically dispatched method calls, where the dynamic types of the objects are unknown.

Outline. The rest of this article is structured as follows. In Section 2 we demonstrate the patterns we use for writing interfaces by providing a small example program that uses interface inheritance and proving that it conforms to its specification. In Section 3 we cover the language and memory-model independent kernel of our Coq formalisation. In Section 4 we specialise our system to handle Java-like programs by providing constructs and a suitable memory model for a subset of Java. Section 5 covers related work, and Section 6 concludes.

2 Reasoning with Interfaces

To demonstrate how our logic is applied, we will use the example of a class Cell that stores a single value and which is extended by a subclass Recell that maintains a backup of the last overwritten value and has an undo operation. This example is originally due to Abadi and Cardelli [1]; a variant of it was also used

[1] The Coq development accompanying this article can be found at `http://itu.dk/people/birkedal/papers/hosl_coq-201105.tar.gz`

```
interface ICell {
  int get();
  void set(int v);
}

class ProxySet {
  static void proxySet(ICell c, int v) {
    c.set(v);
  }
}

class Cell implements ICell {
  int value;

  Cell() { }
  int get() {
    return this.value;
  }
  void set(int v) {
    this.value = v;
  }
}
```

```
interface IRecell extends ICell {
  void undo();
}

class Recell implements IRecell {
  Cell cell;
  int bak;

  Recell() {
    this.cell = new Cell();
  }
  int get() {
    return this.cell.get();
  }
  void set(int v) {
    this.bak = this.cell.get();
    this.cell.set(v);
  }
  void undo() {
    this.cell.set(this.bak);
  }
}
```

Fig. 1. Java code for the Cell-Recell example with interface inheritance

by Parkinson and Bierman [14] to show how their logic deals with class-to-class inheritance.

We add to this example a method proxySet, which calls the set method of a given object reference. It is a challenge to give a single specification to this method that is powerful enough to expose any additional side effects the set method might have in arbitrary subclasses. We will see in this section how our specification style achieves this, and it is sketched in Section 5 how this compares to related work.

Our model programming language is a subset of both Java and $C\sharp$. It leaves out class-to-class inheritance and focuses on interface inheritance. This mode of inheritance captures the essential object-oriented aspect of dynamic dispatch, while the code-reuse aspect has to be explicitly encoded with class composition. A Java implementation of the Cell-Recell example can be found in in Figure 1.

2.1 Interface ICell

Interface ICell from Figure 1 is modelled as a parametrised specification that states conditions for whether a class C behaves "Cell-like". In the following, *val* denotes the type of program values, in our case the union of integers, Booleans and object references. Also, $\mathit{UPred}(\mathit{heap})$ is the type of logical propositions over heaps, i.e., the spatial component of the assertion logic (see Section 3.1 for the precise definition).

$$\begin{aligned} \mathit{ICell} \triangleq\ & \lambda C : \mathit{classname}. \quad \lambda T : \mathit{Type}. \quad \lambda R : \mathit{val} \to T \to \mathit{UPred}(\mathit{heap}). \\ & \lambda g : T \to \mathit{val}. \quad \lambda s : T \to \mathit{val} \to T. \\ & (\forall t : T.\ C{::}\mathsf{get}(\mathsf{this}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\mathsf{r}.\ \widehat{R}\ \mathsf{this}\ t \wedge \mathsf{r} = g\ t\}) \wedge & (1) \\ & (\forall t : T.\ C{::}\mathsf{set}(\mathsf{this}, \mathsf{x}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\widehat{R}\ \mathsf{this}\ (\widehat{s}\ t\ \mathsf{x})\}) \wedge & (2) \\ & (\forall t, v.\ g\ (s\ t\ v) = v) & (3) \end{aligned}$$

There is some notation to explain here. *ICell* is a function that takes five arguments and returns a result of type *spec*, which is the type of specifications. The logical connectives at the outer level ($\wedge$ and $\forall$) thus belong to the specification logic. The parameter R is the representation predicate of class C, so $R\ c\ t$ intuitively means that c is a reference to an object that is mathematically modelled by the value t of type T. The parameters g and s are functions that describe how get and set inspect and transform this mathematical value. They are constrained by (3) to ensure that get will actually return the value set with set.

The notation $C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\}$ from (1) and (2) specifies that method m of class C has precondition P and postcondition Q. The arguments in a call will be bound to the names $\bar{p}$ in P and Q, and the return value will be bound to r in Q. We support both static and dynamic methods, where dynamic methods have an additional first argument, as seen in (1) and (2). The precise definition is given in Section 4.2.

The notation $\widehat{f}$ from (1) and (2) lifts a function f such that it operates on expressions, including program variables, rather than operating directly on *val*. It is a technical point that can be ignored for a first understanding of this example, but it is crucial for making HOSL work in a stack-based language. Details are in Section 3.2.

The type of T refers to the *Type* universe hierarchy in Coq.

2.2 Method Proxyset

Consider method proxySet from Figure 1. Operationally, calling proxySet(c, v) does the same as calling c.set(v), and we seek a specification that reflects this. It is crucial for modularity that proxySet can be specified and verified only once and then used with any implementation of ICell that may be defined later. We give it the following specification.

$$\begin{aligned} \mathit{ProxySet\_spec} \triangleq\ & \forall C, T, R, g, s.\ \mathit{ICell}\ C\ T\ R\ g\ s \to \\ & \forall t : T.\ \mathsf{ProxySet{::}proxySet}(\mathsf{c}, \mathsf{x}) \mapsto \{\mathsf{c} : C \wedge \widehat{R}\ \mathsf{c}\ t\}\_\{\widehat{R}\ \mathsf{c}\ (\widehat{s}\ t\ \mathsf{x})\} \end{aligned}$$

The assertion $\mathsf{c} : C$ means that the object referenced by c is of class C. Thus, the caller of proxySet can pass in an object reference of any class C as long as C can be shown to satisfy *ICell*.

This specification is as powerful as that of set in *ICell* since it essentially forwards it. Any class that behaves Cell-like should be able to encode the behaviour of its set method by a suitable choice of R and s. We will see in Section 2.6 that it, for instance, is possible to pass in a Recell and deduce how proxySet affects its backup value.

2.3 Class Cell

A Java implementation of Cell can be found in Figure 1. We model constructors as static methods that allocate the object before running the initialisation code and return the allocated object, which is what happens in the absence of class-to-class inheritance.

We give class Cell the following specification, which is a conjunction of what we will call an *interface specification* and a *class specification.* These correspond respectively to the *dynamic* and *static* specifications in [14].

$$\mathit{Cell\_spec} \triangleq \exists R_{\mathrm{Cell}}.\ \mathit{ICell}\ \mathsf{Cell}\ \mathit{val}\ R_{\mathrm{Cell}}\ (\lambda v.\ v)\ (\lambda \_, v.\ v) \wedge \mathit{Cell\_class}\ R_{\mathrm{Cell}}$$

where

$$\begin{array}{ll} \mathit{Cell\_class} \triangleq & \lambda R_{\mathrm{Cell}} : \mathit{val} \rightarrow \mathit{val} \rightarrow \mathit{UPred}(\mathit{heap}). \\ & \quad \mathsf{Cell::new}() \mapsto \{\mathit{true}\}\_\{\exists v.\ \widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\} \wedge \\ & \quad (\forall v.\ \mathsf{Cell::get}(\mathsf{this}) \mapsto \{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\}\_\{\mathsf{r}.\ \widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v \wedge \mathsf{r} = v\}) \wedge \\ & \quad (\forall v.\ \mathsf{Cell::set}(\mathsf{this}, \mathsf{x}) \mapsto \{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\}\_\{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ \mathsf{x}\}) \end{array}$$

The representation predicate R_{Cell} is quantified such that its definition is visible only while proving the specifications of Cell, thus hiding the internal representation of the class from clients [4,13].

It is crucial that R_{Cell} is quantified outside both the class and the interface specification such that the representation predicate is the same in the two. In practice, a client will allocate a Cell by calling new, which establishes R_{Cell}; later, to model casting the object reference to its interface type, the client knows that *ICell* holds for this same R_{Cell}.

The specifications of get and set in *Cell_class* are identical to their counterparts in *ICell* when C, T, R, g, and s, are instantiated as in *Cell_spec*. In general, the class specification can be more precise than the interface specification, similarly to the dynamic and static specifications of [14].

To prove *Cell_spec*, the existential R_{Cell} is chosen as $\lambda c, v.\ c.\mathsf{value} \mapsto v$. We can then show that *Cell_class* R_{Cell} holds by verifying the method bodies of get, set and init, and the correctness of get and set can be used as a lemma in proving the interface specification. In this way, each method body is verified only once.

2.4 Interface IRecell

To show the analogy to interface inheritance at the specification level, we examine an interface for classes that behave Recell-like. The Java code for that is IRecell in Figure 1. The specification corresponding to this interface follows the same pattern as *ICell*:

$$\begin{array}{lll} \mathit{IRecell} \triangleq & \lambda C : \mathit{classname}. \quad \lambda T : \mathit{Type}. \quad \lambda R : \mathit{val} \rightarrow T \rightarrow \mathit{UPred}(\mathit{heap}). & \\ & \lambda g : T \rightarrow \mathit{val}. \quad \lambda s : T \rightarrow \mathit{val} \rightarrow T. \quad \lambda u : T \rightarrow T. & \\ & \mathit{ICell}\ C\ T\ R\ g\ s \wedge & (4) \\ & (\forall t : T.\ C\mathsf{::undo}(\mathsf{this}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\widehat{R}\ \mathsf{this}\ (u\ t)\}) \wedge & (5) \\ & (\forall t, v.\ g\ (u\ (s\ t\ v)) = g\ t) & (6) \end{array}$$

Notice that interface extension is modelled by referring to *ICell* in (4). We do not have to respecify get and set since they were already general enough in *ICell* due to it being parametric in g and s. Note how equation (6) specifies the abstract behaviour of undo via g and s.

There is a pattern to how we construct a specification-logic interface predicate from a Java interface declaration. For each method $m(x_1, \ldots, x_n)$, we add a parameter $f_m : T \to \mathit{val}^n \to (\mathit{val} \times T)$. The product $(\mathit{val} \times T)$ can be replaced with just *val* or T if the method should have no side effects or no return value, respectively. We then add a method specification of the form:

$$\forall t : T.\ C{::}m(\bar{p}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\mathsf{r}.\ \widehat{R}\ \mathsf{this}\ (\pi_2\ (\widehat{f_m}\ \bar{p}\ t)) \wedge \mathsf{r} = \pi_1\ (\widehat{f_m}\ \bar{p}\ t)\}.$$

2.5 Class Recell

The specification of class Recell follows the same pattern as with Cell:

$$\begin{array}{rl} \mathit{Recell\_spec} \triangleq & \exists R_{\mathrm{Recell}} : \mathit{val} \to \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}). \\ & \mathit{IRecell}\ \mathsf{Recell}\ (\mathit{val} \times \mathit{val})\ R\ g\ s\ u \wedge \mathit{Recell\_class}\ R_{\mathrm{Recell}} \\ \text{where} & R = \lambda \mathit{this}, (v, b).\ R_{\mathrm{Recell}}\ \mathit{this}\ v\ b, \qquad g = \lambda(v, b).\ v, \\ & s = \lambda(v, b), v'.\ (v', v), \qquad u = \lambda(v, b).\ (b, b), \end{array}$$

and *Recell_class* is defined analogously to *Cell_class*.

2.6 Class World

The correctness of the above specifications only matters if it enables client code to instantiate and use the classes. The client code in World demonstrates this:

```
class World {
  static ICell make() {
    Recell r = new Recell();
    r.set(5);
    ProxySet::proxySet(r, 3);
    r.undo();
    return r;
  }

  static void main() {
    ICell c = World::make();
    assert c.get() == 5;
  }
}
```

The body of make demonstrates the use of proxySet. Operationally, it should be clear that r has the value 3 and the backup value 5 after the call to proxySet. This can also be proved in our logic despite using a specification of proxySet that was verified without knowledge of Recell and its backup field.

Upon returning from make, we choose to forget that the returned object is really a Recell, upcasting it to ICell. Its precise class is not needed by the caller, main, which only needs to know that the returned object will return 5 from get.

We capture the interaction between these two methods with the following specification, in which $\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\mathit{heap})$ injects the specification logic into the logic of propositions over heaps, thus generalising the concept of nested triples. Section 3.5 describes *FunI* in more detail.

$$\begin{aligned}&\mathit{World\_spec} \triangleq \mathsf{World::main()} \mapsto \{\mathit{true}\}\_\{\mathit{true}\} \wedge \\ &\quad \mathsf{World::make()} \mapsto \{\mathit{true}\}\_\left\{\begin{array}{l}\mathsf{r}.\ \exists C, T, R, g, s.\ \widehat{\mathit{FunI}}\ (\mathit{ICell}\ C\ T\ R\ g\ s) \wedge \\ \quad \exists t.\ \widehat{R}\ \mathsf{r}\ t \wedge g\ t = 5 \wedge \mathsf{r} : C\end{array}\right\}\end{aligned}$$

The make method is specified to return an object whose class C is unknown, but we know that C satisfies *ICell*.

This pattern of returning an object of an unknown type that satisfies a particular specification often comes up in object-oriented programming: think of the method on a collection that returns an iterator, for example. The essence of this pattern is to have a parametrised specification $S : \mathit{classname} \rightarrow \mathit{spec}$ and a method specified as $D{::}m() \mapsto \{\mathit{true}\}\_\{\mathsf{r}.\ \exists C.\ \mathsf{r} : C \wedge \widehat{\mathit{FunI}}\ (S\ C)\}$. A more straightforward alternative to such a specification – one that does not require an embedding of the specification logic in the assertion logic – would be $\exists C.\ S\ C \wedge D{::}m() \mapsto \{\mathit{true}\}\_\{\mathsf{r}.\ \mathsf{r} : C\}$. However, this restricts the body of m to only being able to return objects of one class. The method body cannot, for example, choose at run time to return either a C_1 or a C_2, where both C_1 and C_2 satisfy S. We find that the most elegant way to allow the method body to make such a choice is to embed the specification in the postcondition.

Using the notion of validity from Definition 5 in Section 3.4 we can now prove that the whole program will behave according to specification:

Theorem 1. *($\mathit{ProxySet\_spec} \wedge \mathit{Cell\_spec} \wedge \mathit{Recell\_spec} \wedge \mathit{World\_spec}$) is valid.*

3 Abstract Representation

The core of our system is designed to be language independent. To allow for different memory models, we adopt the notion of separation algebras from Calcagno et al. [6]; we can then instantiate an assertion logic with any separation algebra suitable for the problem at hand. Commands are modelled as relations on the program state, which in turn consists of a mutable stack and a heap. Finally, we define an expressive specification logic that can be used to reason about semantic commands.

We use set-theoretic notation to describe our formalisation as this makes the theories easier to read; in Coq we model these sets as functions into *Prop*, which is the sort of propositions in Coq.

3.1 Uniform Predicates

Definition 1 (Separation algebra). *A separation algebra is a partial, cancellative, commutative monoid (Σ, $\circ$, $\mathbf{1}$) where Σ is the carrier, $\circ$ is the monoid operator, and $\mathbf{1}$ is the unit element.*

Intuitively, Σ can be thought of as a type of heaps, and the $\circ$-operator as composition of disjoint heaps. Hence we refer to the elements of Σ as heaps. Two heaps are compatible, written $h_1 \# h_2$ if $h_1 \circ h_2$ is defined. A heap h_1 is a subheap of a h_2, written $h_1 \sqsubseteq h_2$, if there exists an h_3 such that $h_2 = h_1 \circ h_3$. We will commonly refer to a separation algebra by its carrier Σ.

A uniform predicate [5] over a separation algebra is a predicate on heaps and natural numbers; it is upwards closed in the heaps and downwards closed in the natural numbers.

$$\mathit{UPred}(\Sigma) \triangleq \{p \subseteq \Sigma \times \mathbb{N} \mid \forall g, m.\ \forall h \sqsupseteq g.\ \forall n \leq m.\ (g,m) \in p \rightarrow (h,n) \in p)\}$$

The upward closure in heaps ensures that we have an intuitionistic separation logic as is desirable for garbage-collected languages.

The natural numbers are used to connect the uniform predicates with the step-indexed specification logic – this connection will be covered in Section 3.5.

We define the standard connectives for the uniform predicates as in [5]:

$$\begin{array}{rcl} \mathit{true} & \triangleq & \Sigma \times \mathbb{N} \qquad\qquad \mathit{false} \ \triangleq \ \emptyset \\ p \wedge q & \triangleq & p \cap q \qquad\qquad\quad\ \ p \vee q \ \triangleq \ p \cup q \\ \forall x : U.\ f & \triangleq & \bigcap_{x:U} f\ x \qquad\ \exists x : U.\ f \ \triangleq \ \bigcup_{x:U} f\ x \\ p \rightarrow q & \triangleq & \{(h,n) \mid \forall g \sqsupseteq h.\ \forall m \leq n.\ (g,m) \in p \rightarrow (g,m) \in q\} \\ p * q & \triangleq & \{(h_1 \circ h_2, n) \mid h_1 \# h_2 \wedge (h_1,n) \in p \wedge (h_2,n) \in q\} \\ p \mathrel{-\!\!*} q & \triangleq & \{(h,n) \mid \forall m \leq n.\ \forall h_1 \# h.\ (h_1,m) \in p \rightarrow (h \circ h_1, m) \in q\} \end{array}$$

For the quantifiers, U is of type *Type*, i.e. the sort of types in Coq, and f is any Coq function from U to $\mathit{UPred}(\Sigma)$. This allows us to quantify over *any* member of *Type* in Coq.

3.2 Stacks

Stacks are functions from variable names to values: $\mathit{stack} \triangleq \mathit{var} \rightarrow \mathit{val}$.

Two stacks are said to agree on a set V of variables if they assign the same value to all members of V: $s \simeq_V s' \triangleq \forall x \in V.\ s\ x = s'\ x$. In order to define operators that take values from the stack as arguments we introduce the notion of a *stack monad*. This approach is similar to that of Varming and Birkedal [20].

$$\mathit{sm}\ T \triangleq \{(f : \mathit{stack} \rightarrow T,\ V : \mathcal{P}(\mathit{var})) \mid \forall s, s'.\ s \simeq_V s' \rightarrow f\ s = f\ s'\}$$

Intuitively, V is an over-approximation of the free program variables in f. For any $m = (f, V) \in \mathit{sm}\ T$, we write $m\ s$ to mean $f\ s$ and $\mathit{fv}\ m$ to mean V.

Theorem 2. *sm is a monad with return operation* $\lambda x : T.\ ((\lambda\_.\ x), \emptyset)$ *and bind operation* $\lambda m : \mathit{sm}\ T.\ \lambda f : T \rightarrow \mathit{sm}\ U.\ ((\lambda s.\ f\ (m\ s)\ s),\ \mathit{fv}\ m \cup \bigcup_{t \in T} \mathit{fv}\ (f\ t))$.

We use the stack monad to model expressions (which can be evaluated to values using data from the stack), pure assertions (that represent logical propositions that are evaluated without using the heap), and assertions (that represent logical propositions that are evaluated using both the heap and the stack).

$$\mathit{expr} \triangleq \mathit{sm}\ \mathit{val} \qquad \mathit{pure} \triangleq \mathit{sm}\ \mathit{Prop} \qquad \mathit{asn}(\varSigma) \triangleq \mathit{sm}\ \mathit{UPred}(\varSigma)$$

We create an assertion logic by lifting all connectives from $\mathit{UPred}(\varSigma)$ into $\mathit{asn}(\varSigma)$. The definitions and properties of the liftings follow from the fact that *sm* is a monad (Theorem 2).We prove that both the uniform predicates and the assertions model separation logic [3].

Theorem 3. *For any separation algebra $\varSigma$, $\mathit{UPred}(\varSigma)$ and $\mathit{asn}(\varSigma)$ are complete BI-algebras.*

The stack monad is also used for the lifting operator $\widehat{f}$ that was introduced in Section 2.1. The operator takes a function f, and returns a function $\widehat{f}$ where any argument type T that is passed to f is replaced with $\mathit{sm}\ T$, and any return type U with $\mathit{sm}\ U$. As an example, the representation predicate R in the specification for *ICell*, which has type $\mathit{val} \to T \to \mathit{UPred}(\mathit{heap})$, is lifted to $\widehat{R}$ in the assertion-logic formulas of the specification. The resulting type for $\widehat{R}$ is $\mathit{sm}\ \mathit{val} \to \mathit{sm}\ T \to \mathit{sm}\ \mathit{UPred}(\mathit{heap})$, i.e. $\mathit{expr} \to \mathit{sm}\ T \to \mathit{asn}(\mathit{heap})$.

We have to make this lifting explicit in specifications because it restricts how program variables behave under substitution. We have that $(\widehat{f}\ e)[e'/x] = \widehat{f}\ (e[e'/x])$ for any $f : \mathit{val} \to \mathit{UPred}(\varSigma)$, but it is not the case that $(g\ e)[e'/x] = g\ (e[e'/x])$ for any $g : \mathit{expr} \to \mathit{asn}(\varSigma)$ because $g\ e$ may have more free program variables than those appearing in e, whereas $\widehat{f}\ e$ cannot, by construction. To make HOSL useful in a stack-based language, where such substitutions are commonplace, we therefore typically quantify over functions into $\mathit{UPred}(\varSigma)$ that we then lift to $\mathit{asn}(\varSigma)$ where needed.

3.3 Semantic Commands

To obtain a language-independent core, we model commands as indexed relations on program states (each consisting of a stack and a heap) – a semantic command will relate, in a certain number of steps, a state either to another state or to an error. The only requirements we impose on these commands are that they do not relate to anything in zero steps, and that they satisfy a frame property that will allow us to infer a frame-rule for all semantic commands. Intuitively, the semantic commands can be seen as abstractions of rules of a step-indexed big-step operational semantics. More formally, we have the following definitions.

Definition 2 (pre-command). *A* pre-command $\tilde{c}$ *relates an initial state to either a terminal state or the special* **err** *state:*

$$\mathit{precmd} \triangleq \mathcal{P}(\mathit{stack} \times \varSigma \times ((\mathit{stack} \times \varSigma) \uplus \{\mathbf{err}\}) \times \mathbb{N})$$

We write $(s, h, \tilde{c}) \leadsto^n x$ to mean that $(s, h, x, n) \in \tilde{c}$.

Definition 3 (Frame property). *A pre-command $\tilde{c}$ has the frame property in case the following holds. If $(s, h_1, \tilde{c}) \not\rightsquigarrow^n$ **err** and $(s, h_1 \circ h_2, \tilde{c}) \rightsquigarrow^n (s', h')$ then there exists h'_1 such that $h' = h'_1 \circ h_2$ and $(s, h_1, \tilde{c}) \rightsquigarrow^n (s', h'_1)$.*

Definition 4 (Semantic command). *A semantic command satisfies the frame property and does not evaluate to anything in zero steps.*

$$\mathit{semcmd} \triangleq \{\hat{c} \in \mathit{precmd} \mid \hat{c} \textit{ has the frame property} \wedge \forall s, h, x.\ (s, h, \hat{c}) \not\rightsquigarrow^0 x\}$$

To facilitate the encoding of imperative programming languages in our framework, we create the following semantic commands that can be used as building blocks for that purpose. These commands are similar to the ones found in [6].

$$\mathbf{id} \qquad \mathbf{seq}\ \hat{c}_1\ \hat{c}_2 \qquad \hat{c}_1 + \hat{c}_2 \qquad \hat{c}^* \qquad \mathbf{assume}\ P \qquad \mathbf{check}\ P$$

Intuitively, these semantic commands are defined as follows: The **id**-command is the identity command – it does nothing; the **seq**-command executes two commands in sequence; the +-operator nondeterministically executes one of two commands; the *-command executes a command an arbitrary amount of times; the **assume**-command assumes a pure assertion that can be used to prove correctness of future commands; the **check**-command works like the **id**-command as long as a pure assertion can be inferred. Recall that pure assertions are logical formulas that are evaluated without using the heap.

Theorem 4. **id***,* **seq***, +, *,* **assume***, and* **check** *are semantic commands.*

3.4 Specification Logic

With the assertion logic and the semantic commands in place, we can define the specification logic. Semantically, a specification is a downwards-closed set of natural numbers; this allows us to reason about (mutually) recursive programs via step-indexing.

$$\mathit{spec} \triangleq \{S \subseteq \mathbb{N} \mid \forall m, n.\ m \leq n \wedge n \in S \rightarrow m \in S\}$$

The set *spec* is a complete Heyting algebra under the subset ordering, i.e., logical entailment ($\models$) is modelled as subset inclusion. Hence a specification S is *valid* if $S = \mathbb{N}$.

Given assertions P and Q, and semantic command $\hat{c}$, we define a Hoare triple specification:

$$\begin{aligned}\{P\}\hat{c}\{Q\} \triangleq \{n \mid \forall m \leq n.\ \forall k \leq m.\ \forall s, h.\ (h, m) \in P\ s \rightarrow (s, h, \hat{c}) \not\rightsquigarrow^k \mathbf{err}\ \wedge \\ \forall h', s'.\ (s, h, \hat{c}) \rightsquigarrow^k (s', h') \rightarrow (h', m - k) \in Q\ s'\}\end{aligned}$$

A program is proved correct by proving that its specification is valid:

Definition 5. *A specification is valid, written $\models S$, when true $\models S$.*

3.5 Connecting the Assertion Logic with the Specification Logic

We define an embedding of the specification logic into the assertion logic as follows:

$$\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\Sigma) \triangleq \lambda S.\ \Sigma \times S.$$

Lemma 1. *FunI is monotone, preserves implication, and has a left and a right adjoint, when spec and UPred(Σ) are treated as poset categories.*

From the second part of this lemma it follows that *FunI* preserves both finite and infinite conjunctions and disjunctions, which entails that all specification logic connectives are preserved by the translation.

3.6 Recursion

The specification connectives defined in the previous section are not enough for our purposes. When proving a program correct (by proving a formula of the form $\models S$), it is commonplace that the proof of one part of specification in S requires other parts of S – a typical example is recursive method calls, where the specification of the method called must be available in the context during its own verification. To accomplish this, we borrow the *later* operator ($\triangleright$) from Gödel-Löb logic (see [2]).

$$\triangleright S \triangleq \{n + 1 \mid n \in S\} \cup \{0\}$$

This operator can be used via the Löb rule, which allows us to do induction on the step-indexes of the semantic commands.

$$\frac{\Gamma \wedge \triangleright S \models S \qquad 0 \in \Gamma \to 0 \in S}{\Gamma \models S}\ \textsc{Löb}$$

In the inductive case $\triangleright S$ is found on the left hand side of the turnstile and can hence be used to prove S.

4 Instantiation to an Object-Oriented Language

We define a Java-like language with syntax of programs $\mathcal{P}$ shown below. The language is untyped and does not need syntax for interfaces; these exist in the specification logic only.

We use a shallow embedding for expressions, which we denote with e, as shown in Section 3.2.

$$\begin{array}{rcl}
\mathcal{P} & ::= & \mathcal{C}^* \qquad\qquad f \in \text{(field names)} \\
\mathcal{C} & ::= & \textbf{class}\ C\ f^*\ (m(\bar{x})\{c;\ \textbf{return}\ e\})^* \\
c & ::= & x := \textbf{alloc}\ C \mid x := e \mid x := y.f \mid x.f := e \mid x := y.m(\bar{e}) \\
& \mid & x := C{::}m(\bar{e}) \mid \textbf{skip} \mid c_1;\ c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\
& \mid & \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{assert}\ e
\end{array}$$

$$\frac{}{\mathbf{skip} \sim_{\mathsf{sem}} \mathbf{id}}\ \textsc{Skip-Sem} \qquad \frac{c_1 \sim_{\mathsf{sem}} \hat{c_1} \qquad c_2 \sim_{\mathsf{sem}} \hat{c_2}}{c_1;\ c_2 \sim_{\mathsf{sem}} \mathbf{seq}\ \hat{c_1}\ \hat{c_2}}\ \textsc{Seq-Sem}$$

$$\frac{c \sim_{\mathsf{sem}} \hat{c}}{\mathbf{while}\ e\ \mathbf{do}\ c \sim_{\mathsf{sem}} \mathbf{seq}\ (\mathbf{seq}\ (\mathbf{assume}\ e)\ \hat{c})^{*}\ (\mathbf{assume}\ \neg e)}\ \textsc{While-Sem}$$

$$\frac{c_1 \sim_{\mathsf{sem}} \hat{c}_1 \qquad c_2 \sim_{\mathsf{sem}} \hat{c}_2}{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \sim_{\mathsf{sem}} (\mathbf{seq}\ (\mathbf{assume}\ e)\ \hat{c}_1) + (\mathbf{seq}\ (\mathbf{assume}\ \neg e)\ \hat{c}_2)}\ \textsc{If-Sem}$$

Fig. 2. The skip, sequential composition, conditional and loop cases of the semantics relation

In order to provide a concrete instance of the assertion logic, we construct a separation algebra of concrete heaps. The carrier set is $\mathit{heap} \triangleq (\mathit{ptr} \times \mathit{field}) \rightharpoonup_{\mathsf{fin}} \mathit{val}$, with the values defined as the union of integers, Booleans and object references. The partial composition $h_1 \circ h_2$ is defined as $h_1 \cup h_2$ if dom $h_1 \cap$ dom $h_2 = \emptyset$; otherwise the result is undefined. The unit of the algebra is the empty map, *emp*. We denote this separation algebra $(\mathit{heap}, \circ, \mathit{emp})$ with *heap*. The points-to predicate is defined as $v.f{\mapsto}v' \triangleq \{(h, n) \mid h \sqsupseteq [(v, f) \mapsto v']\}$.

4.1 Semantics of the Programming Language

We define the semantics of the programming language commands by relating them to semantic commands instantiated with *heap* as the separation algebra. We write $c \sim_{\mathsf{sem}} \hat{c}$ to denote that the syntactic command c is related to the semantic command $\hat{c}$. The $\sim_{\mathsf{sem}}$ relation can be thought of as a function; it is defined as a relation only because this was more straightforward in Coq.

The commands **skip**, ;, **if**, and **while** can be related directly to composites of the general semantic commands, defined in Section 3.3. The definition of $\sim_{\mathsf{sem}}$ for these commands can be found in Figure 2. For the remaining commands, new semantic commands must be created.

In particular, for method calls, we define a semantic command

$$\mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}$$

that, intuitively, calls method m of class C with arguments $\bar{e}$ and assigns the return value to x; the command c is the method body, and $\hat{c}$ is its corresponding semantic command. This semantic command works uniformly for both static and dynamic methods, since in the dynamic case we can pass the object reference as an additional argument. The definition of this semantic command is shown in Figure 3. The definition makes use of a predicate

$$C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P}$$

which holds in case method m in class C has parameters $\bar{p}$ and method body c in program $\mathcal{P}$. The program parameter $\mathcal{P}$ has been left implicit in the other

$$\frac{([\bar{p} \mapsto (\bar{e}\ s)], h, \hat{c}) \rightsquigarrow^{n} (s', h') \qquad C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| = |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{n+1} (s[x \mapsto (r\ s')], h')}\ \textsc{Call}$$

$$\frac{C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \notin \mathcal{P}}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{1} \mathbf{err}}\ \textsc{Call-Fail1}$$

$$\frac{C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| \neq |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{1} \mathbf{err}}\ \textsc{Call-Fail2}$$

$$\frac{([\bar{p} \mapsto (\bar{e}\ s)], h, \hat{c}) \rightsquigarrow^{n} \mathbf{err} \qquad C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| = |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{n+1} \mathbf{err}}\ \textsc{Call-Fail3}$$

Fig. 3. Semantic call commands

rules. The notation $[\bar{p} \mapsto (\bar{e}\ s)]$ denotes a finite map that associates each p in $\bar{p}$ with the e at the corresponding position in $\bar{e}$ evaluated in stack s.

The requirement that the method body is related to the semantic command is not enforced by the construction of the semantic command, but rather by the definition of $\sim_{\mathsf{sem}}$ for respectively static and dynamic method calls:

$$\frac{c \sim_{\mathsf{sem}} \hat{c}}{x := C{::}m(\bar{e}) \sim_{\mathsf{sem}} \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}}\ \textsc{SCall-Sem}$$

$$\frac{c \sim_{\mathsf{sem}} \hat{c} \qquad y : C}{x := y.m(\bar{e}) \sim_{\mathsf{sem}} \mathbf{call}\ x\ C{::}m(y, \bar{e})\ \mathbf{with}\ c\ \hat{c}}\ \textsc{DCall-Sem}$$

4.2 Syntactic Hoare Triples and the Concrete Assertion Logic

Hoare triples for syntactic commands are defined in the following manner:

$$\{P\}c\{Q\} \triangleq \forall \hat{c}.\ c \sim_{\mathsf{sem}} \hat{c} \rightarrow \{P\}\hat{c}\{Q\}.$$

From this definition we infer and prove sound Hoare rules for all commands of our language. To define the rule for method calls we first define the predicate that asserts the specification of methods, introduced in Section 2.1.

$$C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \triangleq \exists c, e.\ \mathit{wf}(\bar{p}, r, P, Q, c) \wedge C{::}m(\bar{p})\{c; \mathbf{return}\ e\} \in \mathcal{P} \\ \wedge \{P\}c\{Q[e/r]\},$$

where *wf* is a predicate to assert the following static properties: the method parameter names do not clash; the pre- and postcondition do not use any stack variables other than the method parameters and this (the postcondition may also use the return variable); the method body does not modify the values of the method parameters or this.

$$\frac{}{\models \{P\}\mathbf{skip}\{P\}}\ \textsc{Skip} \qquad \frac{}{\{P\}c_1\{Q\} \wedge \{Q\}c_2\{R\} \models \{P\}c_1;\ c_2\{R\}}\ \textsc{Seq}$$

$$\frac{}{\{P \wedge e\}c_1\{Q\} \wedge \{P \wedge \neg e\}c_2\{Q\} \models \{P\}\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{Q\}}\ \textsc{If}$$

$$\frac{}{\{P \wedge e\}c\{P\} \models \{P\}\mathbf{while}\ e\ \mathbf{do}\ c\{P \wedge \neg e\}}\ \textsc{While} \qquad \frac{P \vdash e}{\models \{P\}\mathbf{assert}\ e\{P\}}\ \textsc{Assert}$$

$$\frac{}{\models \{\mathit{true}\}x :=\ \mathbf{alloc}\ C\{\forall^* f \in \mathit{fields}(C).\ x.f \mapsto \mathit{null}\}}\ \textsc{Alloc}$$

$$\frac{}{\models \{P\}x := e\{\exists v.\ P[v/x] \wedge x = e[v/x]\}}\ \textsc{Assign} \quad \frac{}{\models \{x.f \mapsto \_\}x.f := e\{x.f \mapsto e\}}\ \textsc{Write}$$

$$\frac{P \vdash y.f \mapsto e}{\models \{P\}x := y.f\{\exists v.\ P[v/x] \wedge x = e[v/x]\}}\ \textsc{Read}$$

$$\frac{\mathit{\Gamma} \models \triangleright C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |y, \bar{e}|}{\mathit{\Gamma} \models \{y : C \wedge P[y, \bar{e}/\bar{p}]\}x := y.m(\bar{e})\{\exists v.\ Q[x, y[v/x], \bar{e}[v/x]/r, \bar{p}]\}}\ \textsc{DCall}$$

$$\frac{\mathit{\Gamma} \models \triangleright C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |\bar{e}|}{\mathit{\Gamma} \models \{P[\bar{e}/\bar{p}]\}x := C{::}m(\bar{e})\{\exists v.\ Q[x, \bar{e}[v/x]/r, \bar{p}]\}}\ \textsc{SCall}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q}{\{P'\}c\{Q'\} \models \{P\}c\{Q\}}\ \textsc{Consequence} \qquad \frac{\forall x \in \mathit{fv}\ R.\ c\ \mathit{does\ not\ modify}\ x}{\{P\}c\{Q\} \models \{P * R\}c\{Q * R\}}\ \textsc{Frame}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q \qquad \mathit{fv}\ P \subseteq \mathit{fv}\ P' \qquad \mathit{fv}\ Q \subseteq \mathit{fv}\ Q'}{C{::}m(\bar{p}) \mapsto \{P'\}\_\{r.\ Q'\} \models C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\}}\ \textsc{Consequence-MSpec}$$

$$\frac{\mathit{fv}\ R \subseteq \{\mathsf{this}\} \cup \bar{p}}{C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \models C{::}m(\bar{p}) \mapsto \{P * R\}\_\{r.\ Q * R\}}\ \textsc{Frame-MSpec}$$

Fig. 4. Specification logic rules for syntactic Hoare triples

Selected proof rules for syntactic commands are shown in Figure 4. Note the use of the *later* operator (▷) in the method call rule; this means that this method call rule will often be used in connection with the Löb rule.

Theorem 5. *The rules in Figure 4 are sound with respect to the operational semantics.*

5 Related Work

Formalisations of higher-order separation logic have been proposed before, e.g. by Varming and Birkedal [20], who developed an Isabelle/HOL formalisation of HOSL for partial correctness for a simple imperative language with first-order mutually recursive procedures, using a denotational semantics of the programming language, and by Preoteasa [15], who developed a PVS formalisation for total correctness using a predicate-transformer semantics for a similar programming language.

Parkinson and Bierman treated an extended version of the Cell-Recell example in [14], improving upon their earlier work in [13]. Their approach is to tailor the specification logic to build in a form of quantification over families of representation predicates following a fixed pattern determined by the inheritance tree of the program. This construction is known as *abstract predicate families* (APFs).

Where our logic allows quantification over a representation type T, as used in Section 2.1, APFs have a built-in notion of variable-arity predicates to achieve same effect: representation predicates of a subclass can add parameters to the representation predicate they inherit. Class Cell defines a two-parameter representation predicate family Val, which is extended to three arguments in Recell. A Recell r having value 2 and backup field 1 would be asserted as $Val(r, 2, 1)$. This assertion implies $Val(r, 2)$, which in turn implies $\exists b.\, Val(r, 2, b)$ if it is known that r is a Recell. Thus, casting to the two-argument representation predicate that would be necessary for calling $\{\exists v.\, Val(\mathsf{c}, v)\}$ proxySet(c, x) $\{\, Val(\mathsf{c}, \mathsf{x})\}$ will lose any information about the backup field.

The logic of Parkinson and Bierman was extended by van Staden and Calcagno [19] to handle multiple inheritance, abstract classes and controlled leaking of facts about the abstract representation of either a single class or a class hierarchy. Using the latter feature, we observe that their logic can also be used to reason about the example in Section 2, by using parameters g and s to give a precise specification of proxySet. Instead of being functions, g and s would be abstract predicate families whose first argument would be an object reference used only for selecting the correct member of the APF.

Compared to the logics based on abstract predicate families, our logic allows families of not just predicates but also types, functions, class names or any other type that can be quantified over in Coq. This gives us strong typing of logical variables, and all this works without building it into the logic and requiring that quantifications and proofs follow the shape of the inheritance tree.

6 Conclusion and Future Work

We have presented a Coq implementation of a generic framework for higher-order separation logic. In this framework, instantiated with a simple object-oriented language, we have shown how HOSL can be used to reason about interfaces and interface inheritance.

Future work includes developing better support for automation via better use of tactics. Our Coq proofs of example programs are cluttered with manual reordering of the context because we do not yet have tactics to automate this. We also plan to integrate the current tool with an Eclipse front-end that is currently being researched within our project [10]. Moreover, we plan to use the tool for formal verification of interesting data structures from the C5 collection library.

Although it is not necessary for the code we mostly want to verify, proper support for class-to-class inheritance in both the logic and the design pattern would enable more direct comparison with related work. It would also make our Java subset more similar to actual Java.

Acknowledgements. We are grateful for useful discussions with Hannes Mehnert, Matthew Parkinson, Peter Sestoft, Kasper Svendsen, and Jacob Thamsborg. This worked was supported in part by the ToMeSo project, funded by the Danish Research Council.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects, 1st edn. Springer, New York (1996)
2. Appel, A.W., Melliès, P.-A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: Proceedings of POPL (2007)
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. Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J., Yang, H.: Step-indexed kripke models over recursive worlds. In: Proceedings of POPL (2011)
6. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proceedings of LICS, pp. 366–378 (2007)
7. Jensen, J., Birkedal, L., Sestoft, P.: Modular verification of linked lists with views via separation logic. Journal of Object Technology (2011); To Appear Preliminary version in FTfJP 2010, `http://www.itu.dk/people/birkedal/papers/views.pdf`
8. Kokholm, N., Sestoft, P.: The C5 generic collection library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen (2006)
9. Krishnaswami, N.R., Aldrich, J., Birkedal, L., Svendsen, K., Buisse, A.: Design patterns in separation logic. In: Proceedings of TLDI, pp. 105–116 (2009)
10. Mehnert, H.: Kopitiam: Modular incremental interactive full functional static verification of java code. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 518–524. Springer, Heidelberg (2011)
11. 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, Springer, Heidelberg (2007)
12. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
13. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of POPL, pp. 247–258 (2005)
14. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: Proceedings of POPL, pp. 75–86 (2008)
15. Preoteasa, V.: Frame rules for mutually recursive procedures manipulating pointers. Theoretical Computer Science (2009)

16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS, pp. 55–74 (2002)
17. 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)
18. 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)
19. van Staden, S., Calcagno, C.: Reasoning about multiple related abstractions with multistar. In: Proceedings of OOPSLA , pp. 504–519 (2010)
20. Varming, C., Birkedal, L.: Higher-order separation logic in Isabelle/HOLCF. Electr. Notes Theor. Comput. Sci. 218, 371–389 (2008)

Relational Decomposition

Lennart Beringer[⋆]

Department of Computer Science, Princeton University,
35 Olden Street, Princeton NJ 08540
eberinge@cs.princeton.edu

Abstract. We introduce relational decomposition, a technique for formally reducing termination-insensitive relational program logics to unary logics, that is program logics for one-execution properties. Generalizing the approach of self-composition, we develop a notion of interpolants that decompose along the phrase structure, and relate these interpolants to unary and relational predicate transformers. In contrast to previous formalisms, relational decomposition is applicable across heterogeneous pairs of transition systems. We apply our approach to justify variants of Benton's Relational Hoare Logic (RHL) for a language with objects, and present novel rules for relating loops that fail to proceed in lockstep. We also outline applications to noninterference and separation logic.

1 Introduction

Verification formalisms and tools based on Hoare logics are typically designed with one-execution properties in mind: their partial or total correctness interpretation involves a single operational judgement $s \xrightarrow{P} t$.

However, many program properties are relational: they are naturally phrased as statements over pairs of executions $s \xrightarrow{P} t$ and $s' \xrightarrow{P'} t'$, stipulating that the terminal states are in relation S whenever the initial states are in relation R. Examples include "obviously relational" properties such as program transformations or noninterference [35], but also extensional interpretations of type systems and program analyses [13].

$$\begin{tikzcd}
s \arrow[r, dash, "R"] \arrow[d, "P"'] & s' \arrow[d, "P'"] \\
t \arrow[r, dash, "S"'] \arrow[ur, dash, "\phi"] & t'
\end{tikzcd}$$

Fig. 1. Relational decomposition of simulation using witness ϕ

In this article, we present *relational decomposition*, a technique for reducing the verification of relational properties to that of unary ones. We demonstrate our technique by deriving a variant of Benton's Relational Hoare Logic (RHL, [13]) from a unary program logic, demonstrating that efforts invested into the construction of semantic models for unary logics can be harnessed for the justification of relational formalisms. We thus open an avenue for integrating relational logics into foundational stacks of verification formalisms [6,4].

[⋆] This work is funded in part by the Air Force Office of Scientific Research (FA9550-09-1-0138) and the National Science Foundation (CNS-0910448).

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 39–54, 2011.

Relational decomposition reduces the validation of a simulation to the separate verification of unary specifications for the executions $s \xrightarrow{P} t$ and $s' \xrightarrow{P'} t'$. These unary specifications are determined by shared relations ϕ that relate terminal states of (nonprimed) executions to the left with initial states of (primed) executions to the right, in effect witnessing the simulation as indicated in Figure 1. The specification for P then arises from the upper left decomposition triangle (with corners s, s', and t), and the specification for P' arises from the lower right triangle (with corners s', t', and t). Each unary specification universally quantifies over states from the opposite execution.

The present article makes two contributions. First, we present fundamental properties of relational decomposition, in a setting in which the two transition systems involved in a simulation are not necessarily identical. We exploit this flexibility when extending relational decomposition to parametrized simulations, i.e. situations where the relations R and S are parametric over values of some type $\mathcal{Z}$. Second, we present specific relational decompositions of relational program logics, for a concrete language of commands, objects, and loops, thus demonstrating how witness relations may be obtained in a concrete setting. More specifically, we

1. establish soundness of decomposition: any witness ϕ yields unary specifications for the left and right executions that together imply the simulation property (Section 2);
2. establish the formal completeness of decomposition: witnesses exist whenever the simulation property semantically holds. The space of witnesses is characterized by an inclusion property between relational predicate transformers. We present laws that relate these transformers to their unary counterparts (Section 2);
3. derive a termination-insensitive variant of RHL in decomposed style, including novel rules for *dissonant* loops, i.e. loops that do not proceed in lock-step; in particular, proof rules synthesize witnesses in a compositional fashion (Section 3);
4. outline an extension of relational decomposition that deals with parametrized simulations. The resulting logic can be used to justify type systems for noninterference [7] and variants of relational separation logics [40] (Section 4).

All results have been verified using the theorem prover Isabelle/HOL, and the source files are available online [14]. As a consequence, details of most proofs are omitted.

1.1 Related Work

Relational decomposition extends the idea of *self-composition* [11]. For the special case of (termination-insensitive) noninterference [35], self-composition establishes the security of a command[1] C by verifying the one-execution property $\{\sim_L\}\ C; C'\{\sim_L\}$ where C' arises from C by replacing all program variables x in C by fresh copies x', and the predicate $\sim_L$ is defined as $\{s \mid \forall x \in L.\ s\ x = s\ x'\}$ for some fixed ("low") subset L of (nonprimed) variables. Self-composition thus reduces relational to unary verification using *syntactic* operations on programs: variable renaming, code duplication, and (sequential) composition. Relational decomposition reveals that the essence

[1] Anticipating the concrete programming language used later in the paper, we let C range over some concrete category of commands, in contrast to the generic labels P.

of self-composition lies neither in the "self" nor in the "composition" aspect, but in the dual use of auxiliary state: related programs do not have to be copies of each other (in fact they need not be syntactically similar at all and may stem from different languages), and no syntactic composition operator is required at their point of interaction. Indeed, the witness relations ϕ can be interpreted as specifications applicable at the point of program composition in a self-composed program, mediating between pre- and postrelations in a style reminiscent of interpolants [17].

Terauchi and Aiken [37] observe that the efficiency of self-composition is improved if phrase-duplication is applied only to small program fragments, but limit their attention largely to noninterference. They demonstrate that type systems for noninterference yield transformation rules that push self-composition towards the leaves of the syntax tree so that the symmetry between C and C' can be better exploited. Our application of relational decomposition is phrased in the opposite direction: we derive a relational logic from a unary one rather than aiming to obtain unary specifications from a given relational specification. The language considered by Terauchi and Aiken [37] is that of simple assignments and while-loops. In particular, heap structures – whose treatment presents a particular challenge due to the fact that differences in location chosen by the allocator in different runs are generally considered unobservable – are not considered.

Naumann [31] extends Terauchi and Aiken's work to a language with objects, for general relational pre- and postconditions. Indistinguishability of locations is treated using the well-known technique of partial bijections [12,7]. Naumann's encoding of relational into unary specifications employs ghost fields: each object contains a boolean ghost field indicating whether the object should be interpreted w.r.t. the left or the right execution, and a further ghost field that (if nonnull) refers to the object's "mate" in the opposite execution. From a semantic point of view this encoding is slightly unsatisfactory, as the soundness result is contingent on the condition that *None of the considered relations or programs should depend on these fields except through explicit use in the encoding* ([31], page 9). Arguably, this condition represents an external assumption whose impact on the end-to-end guarantee is not formally modeled, requiring the end-user to trust some additional tool validating (possibly a syntactic approximation of) this condition. In fact, independence is itself a relational concept – and so is arguably the concept of ghost variables: the rules governing their use are virtually identical to those for a high-security variable in noninterference. A practical drawback of Naumann's encoding is that the explicit declaration of ghost fields permeates all classes, potentially limiting the scalability of the approach ([31], page 16).

Beringer-Hofmann [15] and Darvas-Hähnle-Sands [18] formulate self-composition in terms of program logics, but again focus on noninterference. In particular, Beringer and Hofmann [15] show how standard type systems [38,24] can be formally interpreted in a unary logic, using a type-directed rule-by-rule construction of intermediate formulae ϕ. The witness relations employed in the present paper extend this construction to arbitrary relational simulation properties over possibly distinct transition systems. The present paper highlights that the synthesis of the witnesses proceeds along the phrase structure or the structure of the RHL proof rules, independent of the type structure.

The logics of Benton [13] and Yang [40] provide a blueprint for our relational Hoare logic, but do not support verification across different languages. These logics are justified by direct recourse to operational semantics rather than being derived from an intermediate unary verification formalism. A further difference consists in our use of a termination-insensitive interpretation of relational judgements: simulations are vacuously fulfilled if either execution fails to terminate. This design decision is motivated by the fact that already in the unary setting, proof techniques for termination (appropriate measures, i.e. variants) are significantly different from those for partial-correctness properties (invariants). A second reason is that applications such as compiler verification often actually relax termination-sensitivity to at least an asymmetric form [28]. Thus, termination appears sufficiently orthogonal to the functional aspects of relational behaviour to be treated separately. Nevertheless, we acknowledge that (and point out where) our design decision has repercussions on the proof rules we are able to derive.

In the area of translation validation, a number of verification approaches have been proposed, some of which include rules for relating loops that fail to proceed in lockstep [32,20,39]. In contrast to our proof system, these approaches are typically justified with the help of auxiliary constructs such as program labels and paths, in conflict with the extensional view taken in the present paper and also emphasized by Benton.

2 The Principle of Decomposition

2.1 Introducing Interpolating Witnesses

For the purpose of this paper, a transition system $\mathcal{T}$ over state space $\mathcal{S}$ and labels $\mathcal{P}$ is a ternary relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$. Contrary to other uses of transition systems, we employ a big-step reading where labels may represent compound program phrases whose cumulative effect is captured in a single transition.

Each transition system $\mathcal{T}$ gives rise to a one-execution specification system where assertions are (curried) binary predicates A over $\mathcal{S}$ that relate initial and final states, similar to postconditions in VDM [25]. We interpret specifications as partial-correctness statements, by writing $\models^{\mathcal{T}} P : A$ whenever $(s, P, t) \in \mathcal{T}$ implies $A\ s\ t$ for all $s, t \in \mathcal{S}$.

The formal notion of simulation employs pre- and postconditions that relate states across two transition systems. In order to clearly distinguish between one- and two-execution specifications we write relational assertions in uncurried, often infix style.

Definition 1. *For $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$ and $\mathcal{T}' \subseteq \mathcal{S}' \times \mathcal{P}' \times \mathcal{S}'$, let $R, S \subseteq \mathcal{S} \times \mathcal{S}'$. Programs $P \in \mathcal{P}$ and $P' \in \mathcal{P}'$ are $R \Longrightarrow S$-similar, notation $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$, if for all s, s', t, and t' with $(s, P, t) \in \mathcal{T}$ and $(s', P', t') \in \mathcal{T}'$, sRs' implies tSt'.*

Unless explicitly remarked otherwise, we follow the convention that nonprimed entities (states, phrases,...) refer to $\mathcal{T}$ and primed entities to $\mathcal{T}'$.

Properties of this shape for $\mathcal{S} = \mathcal{S}'$ include determinism (choose R and S to be equality), liveness of variables (choose R and S to be equality on live-in variables) and slicing, intra-language transformations, and termination-insensitive versions of properties considered by [13]. Further instances arise when the condition $\mathcal{S} = \mathcal{S}'$ is dropped, including variations of compiler correctness, refinement, and abstract interpretation.

The core of relational decomposition consists of the operators Dec_{L} and Dec_{R}

$$Dec_{\mathsf{L}}\ R\ \phi\ s\ t = \forall s'.\ sRs' \rightarrow t\phi s' \text{ and } Dec_{\mathsf{R}}\ S\ \phi\ s'\ t' = \forall t.\ t\phi s' \rightarrow tSt'. \qquad (1)$$

Given $\phi \subseteq \mathcal{S} \times \mathcal{S}'$, operator $\mathit{Dec}_{\mathsf{L}}$ constructs a unary assertion for the left decomposition triangle from Figure 1, i.e. for the execution of P. In fact, the construction uniformly applies for all types $\mathcal{S}'$ subject to $R \subseteq \mathcal{S} \times \mathcal{S}'$. Similarly, $\mathit{Dec}_{\mathsf{R}}$ constructs a unary assertion for the right decomposition triangle from Figure 1, the execution of P', uniformly for types $\mathcal{S}$ with $S \subseteq \mathcal{S} \times \mathcal{S}'$. The operators are motivated by the following result.

Lemma 1. (Soundness) *Suppose* $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\, R\, \phi$ *and* $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\, S\, \phi$*. Then* $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$*.*

Thus, the task of verifying $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$ is reduced to the task of exhibiting an arbitrary ϕ that satisfies the two one-execution properties. Each suitable witness ϕ is a (relational) interpolant between the relational precondition R and the relational postcondition S, by virtue of constraints (1). Before deriving concrete interpolants that justify RHL in Section 3, we discuss further formal properties of decomposition.

2.2 Properties of Decomposition Operators

We first observe that $\mathit{Dec}_{\mathsf{L}}$ is covariant in ϕ and contravariant in R, i.e. for $\phi \subseteq \psi$ and $Q \subseteq R$, $\mathit{Dec}_{\mathsf{L}}\; R\; \phi\; s\; t$ implies $\mathit{Dec}_{\mathsf{L}}\; Q\; \psi\; s\; t$, while $\mathit{Dec}_{\mathsf{R}}$ is covariant in S and contravariant in ϕ, i.e. for $\psi \subseteq \phi$ and $S \subseteq T$, $\mathit{Dec}_{\mathsf{R}}\; S\; \phi\; s'\; t'$ implies $\mathit{Dec}_{\mathsf{R}}\; T\; \psi\; s'\; t'$. We also note the identity $\mathit{Dec}_{\mathsf{R}}\; S\; \phi = \mathit{Dec}_{\mathsf{L}}\; \phi^{-1}\; S^{-1}$. Next, we characterize the witnesses suitable for establishing Lemma 1. To this end, consider the operators

$$\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; R = \{(t, s') \mid \exists s.\ (s, P, t) \in \mathcal{T} \wedge sRs'\}$$
$$\phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; S = \{(t, s') \mid \forall t'.\ (s', P', t') \in \mathcal{T}' \to tSt'\}$$

The former constructs a candidate for ϕ according to the upper left triangle in the diagram, given R and P. The latter constructs a (in general different) candidate ϕ according to the lower right triangle in the diagram, given S and P'. In point-free notation [22], $\phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; S$ can be written as $\widehat{P'}\backslash S$ where $\widehat{P'}$ denotes the uncurried form of the transition relation for P' and the *weakest prespecification* $X\backslash Y$ is defined as $\overline{\overline{Y}; X^{-1}}$.

By construction, these operators are covariant in their second argument and yield valid specifications for their defining triangles:

Lemma 2. *We have* $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\; R\; (\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; R)$ *and* $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\; S\; (\phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; S)$*.*

They also satisfy $\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; (\phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; T) \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; (\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; T)$ and set-theoretic laws such as $\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; (R \cap T) \subseteq \phi^{\mathcal{T}}_{\mathsf{L}}\; P\; R \cap \phi^{\mathcal{T}}_{\mathsf{L}}\; P\; T$. In particular, $\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; R$ is the *least* relation obeying the left triangle, and $\phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; S$ is the *greatest* relation obeying the right triangle:

Lemma 3. *If* $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\; R\; \phi$ *then* $\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; R \subseteq \phi$*. If* $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\; S\; \phi$ *then* $\phi \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; S$*.*

Thus, any witness ϕ from Lemma 1 is sandwiched between the two operators, i.e. satisfies $\phi^{\mathcal{T}}_{\mathsf{L}}\; P\; R \subseteq \phi \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\; P'\; S$. Conversely, either operator is suitable as a witness:

Lemma 4. (Completeness) *Suppose* $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$. *Then*

1. $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$ *and* $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$
2. $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$ *and* $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$.

Combining the above lemmas, we obtain the following.

Theorem 1. $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$ *iff* $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$.

Proof. For the implication from left to right, we apply Lemma 4(1) to obtain $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$. Then, Lemma 3 (part 2) yields $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$. For the opposite implication, we have $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$ by Lemma 2, so $\models^{\mathcal{T}} P : \mathit{Dec}_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$ by covariance. We also have $\models^{\mathcal{T}'} P' : \mathit{Dec}_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$ (again by Lemma 2), hence the result follows by applying Lemma 1 to $\phi := \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$.

The operators are defined from the relational perspective, but are also intimately connected with the unary transformers

$$\begin{aligned} &\mathit{Strongest\ postcondition} : \mathit{SP}^{\mathcal{T}}_{P}(X) = \{t \mid \exists\, s \in X.\ (s, P, t) \in \mathcal{T}\} \\ &\mathit{Weakest\ lib.\ precondition} : \mathit{WLP}^{\mathcal{T}'}_{P'}(Y') = \{s' \mid \forall\, t'.\ (s', P', t') \in \mathcal{T}' \to t' \in Y'\} \end{aligned}$$

where X and Y' are state sets from $\mathcal{T}$ and $\mathcal{T}'$, respectively. Indeed, we have

$$\begin{aligned} \phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R &= \{(t, s') \mid t \in \mathit{SP}^{\mathcal{T}}_{P}(\{s \mid sRs'\})\} \\ \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S &= \{(t, s') \mid s' \in \mathit{WLP}^{\mathcal{T}'}_{P'}(\{t' \mid tSt')\})\} \end{aligned} \tag{2}$$

Substituting these equalities into Theorem 1, we have that $R \Longrightarrow S$-similarity is soundly and completely characterized by the inclusion of the left SP in the right WLP:

$$\{(t, s') \mid t \in \mathit{SP}^{\mathcal{T}}_{P}(\{s \mid sRs'\})\} \subseteq \{(t, s') \mid s' \in \mathit{WLP}^{\mathcal{T}'}_{P'}(\{t' \mid tSt'\})\}. \tag{3}$$

We may also define the relational transformers

$$\begin{aligned} &\mathit{Strongest\ postrelation} : \\ &\quad \mathit{SR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(R) = \{(t, t') \mid \exists\, s\, s'.\ (s, P, t) \in \mathcal{T} \wedge (s', P', t') \in \mathcal{T}' \wedge sRs'\} \\ &\mathit{Weakest\ lib.\ prerelation} : \\ &\quad \mathit{WLR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(S) = \{(s, s') \mid \forall\, t\, t'.\ (s, P, t) \in \mathcal{T} \to (s', P', t') \in \mathcal{T}' \to tSt'\}. \end{aligned}$$

These satisfy the following properties.

Lemma 5. *We have*

1. $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ (\mathit{WLR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(S)) \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$
2. $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ (\mathit{SR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(S))$
3. $\begin{aligned}[t] \mathit{WLR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(S) &= \{(s, s') \mid s' \in \mathit{WLP}^{\mathcal{T}'}_{P'}(\{t' \mid s \in \mathit{WLP}^{\mathcal{T}}_{P}(\{t \mid tSt'\})\})\} \\ &= \{(s, s') \mid s \in \mathit{WLP}^{\mathcal{T}}_{P}(\{t \mid s' \in \mathit{WLP}^{\mathcal{T}'}_{P'}(\{t' \mid tSt'\})\})\} \end{aligned}$
4. $\begin{aligned}[t] \mathit{SR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(R) &= \{(t, t') \mid t' \in \mathit{SP}^{\mathcal{T}'}_{P'}(\{s' \mid t \in \mathit{SP}^{\mathcal{T}}_{P}(\{s \mid sRs'\})\})\} \\ &= \{(t, t') \mid t \in \mathit{SP}^{\mathcal{T}}_{P}(\{s \mid t' \in \mathit{SP}^{\mathcal{T}'}_{P'}(\{s' \mid sRs'\})\})\}. \end{aligned}$

The latter two equations show that $R \Longrightarrow S$-similarity can also be verified by sequentially applying the respective unary liberal precondition operators (item 3) and verifying $R \subseteq \mathit{WLR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(S)$, or by applying the respective unary strongest postcondition operators (item 4) and verifying $\mathit{SR}^{\mathcal{T},\mathcal{T}'}_{P,P'}(R) \subseteq S$. The mixed positive and negative occurrences of interpolants in the definition of $\mathit{Dec}_{\mathsf{L}}$ and $\mathit{Dec}_{\mathsf{R}}$ highlight that interpolants capture the property applicable at the "point of composition" in self-composition, i.e. at the state where P has executed but P' has not started yet, as captured by equation (3).

3 Application: Decomposed Justification of Relational Hoare Logic

We now instantiate the generic development to the situation where $\mathcal{T}$ and $\mathcal{T}'$ coincide and are equal to the operational judgement of an imperative language with objects.

3.1 Language Definition and Unary Program Logic

We assume infinite and distinct categories of variables $x, y \in \mathcal{X}$, field identifiers $f \in \mathcal{F}$, class identifiers $c \in \mathcal{C}$, and locations $\ell \in \mathcal{L}$. The space of finite partial functions from A to B is denoted by $A \rightharpoonup B$, and the space of total functions by $A \Rightarrow B$. Operations and constructions such as update, domain, and range, are defined and denoted in the standard fashion. A value $v \in \mathcal{V}$ is either an integer value i, a location ℓ, or Null. Value expressions $e \in \mathcal{E}$ are value constants, variables, or binary operators, while boolean expressions b are binary predicates over values. The syntax of commands is

$$\begin{aligned} C \in \mathcal{P} ::= {} & \mathbf{Skip} \mid x{:=}e \mid x := \mathbf{new}\ c\ \iota \mid x{:=}y.f \mid x.f{:=}e \mid \\ & C;D \mid \mathbf{While}\ b\ \mathbf{do}\ C \mid \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \end{aligned}$$

where $\iota \in \mathcal{F} \rightharpoonup \mathcal{E}$ specifies the initialization of fields in the absence of a formalized class system.

The operational semantics is defined over objects, heaps, stores, and states

$$\begin{array}{ll} o \in \mathcal{O} \equiv \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}) & \quad s \in \mathcal{R} \equiv \mathcal{X} \Rightarrow \mathcal{V} \\ h \in \mathcal{H} \equiv \mathcal{L} \rightharpoonup \mathcal{O} & \quad \sigma \in \Sigma \equiv \mathcal{R} \times \mathcal{H} \end{array}$$

[2]We write $[\![e]\!]_s$ and $[\![b]\!]_s$ for the (heap-independent) evaluation of value and boolean expressions, respectively, and map the former operation over initialization maps in the expected manner.

The transition system $\mathcal{T}_{\mathsf{Obj}} \subseteq \Sigma \times \mathcal{P} \times \Sigma$, with pretty-printed judgements $\sigma \xrightarrow{C} \tau$, is defined as a big-step relation, with nondeterministic allocation

$$\textsc{OpNew}\frac{\ell \notin \mathit{locs}\ (s, h)}{(s, h) \xrightarrow{x:=\mathbf{new}\ c\ \iota} (s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)])}$$

[2] The use of $s, t, \ldots$ for concrete stores as well as for states of abstract transition systems should not lead to confusion, as instantiations to the concrete language are always discussed separately from the abstract treatment.

($locs\ \sigma$ denotes the set of all locations ℓ occurring in σ) and field modification rule

$$\textsc{OpPut}\,\frac{s\,x = \ell \quad h\,\ell = (c, F)}{(s,h) \xrightarrow{x.f:=e} (s, h[\ell \mapsto (c, F[f \mapsto [\![e]\!]_s])])}.$$

The semantics does not model error states or stuck executions explicitly: attempts to access dangling pointers, Null, or undefined fields of allocated objects result in the absence of a formal derivation.

In accordance with the setup of Section 2.1, we have derived a unary logic with judgements of the form $\rhd C : A$ where A are curried relations over Σ. The proof rules are essentially those given in [15], plus rules for object allocation

$$\frac{}{\rhd x := \mathbf{new}\ c\ \iota : \lambda\ (s,h)\ \tau.\ \exists\ \ell \notin locs\ (s,h).\ \tau = (s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)])}$$

and for the field accessing instructions (omitted). Using standard techniques [26,33], we have proven the logic sound and complete, relative to the ambient logic HOL:

Theorem 2. *$\rhd C : A$ holds if and only if $\models^{\mathcal{T}_{\mathsf{Obj}}} C : A$.*

3.2 Derivation of Relational Proof Rules

Instantiating $\mathcal{T} = \mathcal{T}_{\mathsf{Obj}}$ and/or $\mathcal{T}' = \mathcal{T}_{\mathsf{Obj}}$ yields laws that decompose the operators $\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R$ and $\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C'\ S$ along the phrase structure, in accordance with the characterizing equations (2). Examples for such laws are

$$\begin{gathered}\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C;D\ R = \phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ D\ (\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R)\\ \phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C';D'\ S = \phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C'\ (\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ D'\ S)\\ \mathit{WLR}_{C;D,C';D'}^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}(S) = \mathit{WLR}_{C,C'}^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}(\mathit{WLR}_{D,D'}^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}(S))\end{gathered}$$

where in the first two cases, the type of the opposite transition system is only constrained by the type of the relations R and S.

Instantiating both transition systems with $\mathcal{T}_{\mathsf{Obj}}$, we now derive proof rules for judgements $C \sim C' : R \Longrightarrow S$. In contrast to Benton [13], but in accordance with Definition 1, we interpret these in the termination-insensitive style. By virtue of the previous section, several formal interpretations of these judgements are compatible with this reading. The derivability from the unary program logic is most explicit if we define $C \sim C' : R \Longrightarrow S$ to be a shorthand for

$$\exists \phi.\ \rhd C : \mathit{Dec}_{\mathsf{L}}\ R\ \phi \wedge \rhd C' : \mathit{Dec}_{\mathsf{R}}\ S\ \phi \tag{4}$$

and then establish the proof rules as derived lemmas. Figure 2 shows selected proof rules for pairs of structurally identical phrases, namely the rule for related object allocations (representative of all rules for relating pairs of atomic instructions) and rules for compound phrases. These rules are similar to the rules given (for the heap-free fragment of the language) by Benton [13]. As is the case in loc. cit., the loop rule is restricted to situations where both iterations proceed in lock-step. Our rule for conditionals allows the executions to proceed along different control paths and consequently has hypotheses for all four possible combinations of branch outcomes.

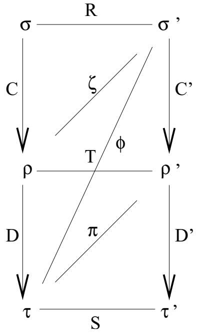
$$\textsc{RhlNew}\ \frac{R = WLR^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}_{x:=\mathbf{new}\ c\ \iota,x':=\mathbf{new}\ c'\ \iota'}(S)}{x := \mathbf{new}\ c\,\iota \sim x' := \mathbf{new}\ c'\ \iota' : R \Longrightarrow S}$$

$$\textsc{RhlComp}\ \frac{C \sim C' : R \Longrightarrow T \qquad D \sim D' : T \Longrightarrow S}{C; D \sim C'; D' : R \Longrightarrow S}$$

$$\textsc{RhlIff}\ \frac{\begin{array}{c} C \sim C' : R \cap \{((s,h),(s',h')) \mid [\![b]\!]_s \wedge [\![b']\!]_{s'}\} \Longrightarrow S \\ D \sim D' : R \cap \{((s,h),(s',h')) \mid \neg[\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \Longrightarrow S \\ C \sim D' : R \cap \{((s,h),(s',h')) \mid [\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \Longrightarrow S \\ D \sim C' : R \cap \{((s,h),(s',h')) \mid \neg[\![b]\!]_s \wedge [\![b']\!]_{s'}\} \Longrightarrow S \end{array}}{\mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \sim \mathbf{If}\ b'\ \mathbf{then}\ C'\ \mathbf{else}\ D' : R \Longrightarrow S}$$

$$\textsc{RhlWhl}\ \frac{\begin{array}{c} C \sim C' : U \Longrightarrow R \qquad R = T \cap \{((s,h),(s',h')).\ [\![b]\!]_s = [\![b']\!]_{s'}\} \\ U = R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \qquad S = R \cap \{((s,h),(s',h')).\ \neg[\![b]\!]_s\} \end{array}}{\mathbf{While}\ b\ \mathbf{do}\ C \sim \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Longrightarrow S}$$

Fig. 2. RHL rules for identically shaped phrases (excerpt)



The derivation of the rules exhibits witnesses as mandated by equation (4). By the results of the previous section, witnesses for the atomic instruction forms may be chosen as $\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R$ or $\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C'\ S$, or any relation sandwiched between the two. Witnesses for compound phrases are synthesized from the witnesses of the constituents, generalizing the noninterference-specific construction from [15]. For example, the witness for the conclusion of rule RhlComp is given by $\phi = \{(\tau, \sigma').\ \exists\, \rho.\ \rho\zeta\sigma' \wedge (\forall\, \rho'.\ \rho T \rho' \rightarrow \tau\pi\rho')\}$ where ζ and π denote the witnesses of the hypotheses, as illustrated on the right. The witness for while rule, $\Phi^{\mathtt{While}}_{(b',R,\phi)}$, is constructed as the least fixed point of the functional

$$\psi \mapsto \left\{ (\tau,(t',k')) \;\middle|\; \begin{array}{l} ([\![b']\!]_{t'} \rightarrow (\exists\, \sigma.\ \sigma\phi(t',k') \wedge (\forall\, \sigma'.\ \sigma R \sigma' \rightarrow \tau\psi\sigma'))) \\ \wedge\ (\neg[\![b']\!]_{t'} \rightarrow \tau R(t',k')) \end{array} \right\}$$

(which is monotone in ϕ and ψ), where ϕ is the witness of $C \sim C' : U \Longrightarrow R$. As a by-product of our generalization, the proofs for the compound phrases reveal a discipline that is not apparent in our earlier noninterference-specific formulation [15]: proofs of the $Dec_{\mathsf{L}}\ .\ .$-conjuncts only use $Dec_{\mathsf{L}}\ .\ .$-clauses of the hypotheses, and proofs of the $Dec_{\mathsf{R}}\ .\ .$-conjuncts only use $Dec_{\mathsf{R}}\ .\ .$-clauses. Thus, the proof system separates into subsystems with specifications $Dec_{\mathsf{L}}\ .\ .$ and $Dec_{\mathsf{R}}\ .\ .$.

In addition to the rules in Figure 2, we have derived rules where the two phrases may be of different shape, including Benton's rules of falsity, consequence, common branch elimination, and dead code elimination – see Figure 3. Carrying a unary judgement in the hypothesis, the dead-code rule applies to arbitrary phrases C whereas Benton only considers the specializations for assignment and while. Conclusions of DeadL may be promoted to phrase compositions using CompSkip. We omit the similar rules for handling dead code and common branches in phrases to the right of $\sim$.

Rule Unary injects a pair of unary judgements into the relational world. This rule is unsound in the termination-sensitive setting. On the other hand, Benton's rule of transitivity $\dfrac{C \sim C' : R \Longrightarrow S \quad C' \sim C'' : R \Longrightarrow S \qquad PER(R \Rightarrow S)}{C \sim C'' : R \Longrightarrow S}$ where $PER(R \Rightarrow S)$ indicates that the function space $R \Rightarrow S$ is a partial equivalence relation[3], is unsound in the termination-insensitive setting: the hypotheses are vacuously satisfied if C' diverges but C and C'' converge. As decomposition witnesses orientate the simulation relation, Benton's rule of symmetry $\dfrac{C \sim C' : R \Longrightarrow S \qquad PER(R \Rightarrow S)}{C' \sim C : R^{-1} \Longrightarrow S^{-1}}$ can be derived if we exploit the semantic symmetry of the simulation relation and use the formal completeness of the program logic, i.e. the reverse direction of Theorem 2. An alternative is to modify the interpretation of judgements, by conjoining (4) with

$$\exists \psi.\ \rhd\ C' : \mathit{Dec}_{\mathsf{L}}\ R^{-1}\ \psi \wedge \rhd\ C : \mathit{Dec}_{\mathsf{R}}\ S^{-1}\ \psi. \tag{5}$$

The resulting interpretation is immediately symmetric and also allows the derivation of the rules above, except for transitivity.

$$\textsc{ComBrL}\ \frac{\begin{array}{c}C \sim C' : U \Longrightarrow S \qquad U = R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \\ D \sim C' : T \Longrightarrow S \qquad T = R \cap \{((s,,h),(s',h')).\ \neg[\![b]\!]_s\}\end{array}}{\textbf{If}\ b\ \textbf{then}\ C\ \textbf{else}\ D \sim C' : R \Longrightarrow S}$$

$$\textsc{DeadL}\ \frac{\rhd\ C : \mathit{Dec}_{\mathsf{L}}\ R\ S}{C \sim \textbf{Skip} : R \Longrightarrow S} \qquad \textsc{CompSkip}\ \frac{C \sim \textbf{Skip} : R \Longrightarrow T \quad D \sim D' : T \Longrightarrow S}{C;D \sim D' : R \Longrightarrow S}$$

$$\textsc{False}\frac{}{C \sim C' : \emptyset \Longrightarrow S} \qquad \textsc{Unary}\ \frac{\begin{array}{c}\rhd\ C : A \qquad \rhd\ C' : A' \\ R = \{(\sigma,\sigma').\ \forall\, \tau\ \tau'.\ A\ \sigma\ \tau \rightarrow A'\ \sigma'\ \tau' \rightarrow \tau S \tau'\}\end{array}}{C \sim C' : R \Longrightarrow S}$$

$$\textsc{Sub}\ \frac{\begin{array}{c}C \sim C' : R \Longrightarrow S \\ R' \subseteq R \quad S \subseteq S'\end{array}}{C \sim C' : R' \Longrightarrow S'} \qquad \textsc{SetOp}\ \frac{\begin{array}{c}C \sim C' : R \Longrightarrow S \quad R' = R \odot T \\ C \sim C' : T \Longrightarrow U \quad S' = S \odot U \quad \odot \in \{\cup,\cap\}\end{array}}{C \sim C' : R' \Longrightarrow S'}$$

Fig. 3. Nonsynchronous RHL rules (excerpt)

Completeness is also used when deriving rules that contain conclusions with phrases that are subphrases of phrases in hypotheses, thus reversing the standard subphrase orientation that is obeyed by our unary logic. For example, the proof of the **Skip**-elimination rule $\dfrac{\textbf{Skip};C \sim C' : R \Longrightarrow S}{C \sim C' : R \Longrightarrow S}$ employs completeness to deduce $\rhd\ C : \mathit{Dec}_{\mathsf{L}}\ R\ \phi$ from $\rhd\ \textbf{Skip};C : \mathit{Dec}_{\mathsf{L}}\ R\ \phi$. An alternative to the use of the formal completeness result would be to work directly at the level of semantic validity, i.e. replace all judgements of the form $\rhd\ C : A$ in (4) or (5) by $\models^{\mathcal{T}_{\mathsf{Obj}}} C : A$.

Theorem 3. *The rules in Figures 2 and 3 are derivable as discussed and thus sound with respect to Definition 1.*

[3] In Benton's setting $R \Rightarrow S$ and $R \Longrightarrow S$ coincide.

3.3 New Rules for Dissonant Loops

Like the rules of Benton [13] and Yang [40], rule RHLWHL from Figure 2 requires the iterations to proceed in lock-step. We have derived two novel rules that overcome this limitation. Our first rule requires both bodies to preserve the invariant individually, decoupling the loops based on a similar motivation as the dead code rules:

$$\frac{\begin{array}{c} C \sim \mathbf{Skip} : (R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\}) \Longrightarrow R \\ \mathbf{Skip} \sim C' : (R \cap \{((s,h),(s',h')).\ [\![b']\!]_{s'}\}) \Longrightarrow R \\ S = R \cap \{((s,h),(s',h')).\ \neg[\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \end{array}}{\mathbf{While}\ b\ \mathbf{do}\ C \sim \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Longrightarrow S}$$

The second rule splits the invariant into preconditions appropriate for synchronized iterations and autonomous iterations.

$$\frac{\begin{array}{ll} C \sim C' : U \Longrightarrow R & U \subseteq R \cap \{((s,h),(s',h')).\ [\![b]\!]_s \wedge [\![b']\!]_{s'}\} \\ C \sim \mathbf{Skip} : V \Longrightarrow R & V \subseteq R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \\ \mathbf{Skip} \sim C' : W \Longrightarrow R & W \subseteq R \cap \{((s,h),(s',h')).\ [\![b']\!]_{s'}\} \\ R \subseteq U \cup V \cup W \cup S & S = R \cap \{((s,h),(s',h')).\ \neg[\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \\ W \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \subseteq U & V \cap \{((s,h),(s',h')).\ [\![b']\!]_{s'}\} \subseteq U \end{array}}{\mathbf{While}\ b\ \mathbf{do}\ C \sim \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Longrightarrow S}$$

This rule is interderivable with the variant where the last two side-conditions (the inclusions $\ldots \subseteq U$) are omitted, for the price of replacing $U \subseteq \ldots$ by $U = \ldots$ in the fist side condition. The earlier rule RHLWHL arises from this variant by setting $V = W = \emptyset$.

The decomposed derivation of the new loop rules employs fixed-point-interpolants similar to $\Phi^{\mathtt{While}}_{(b',R,\phi)}$ above. For the details, see [14].

As an example for the application of these rules, consider the programs

$$C \equiv r{:=}0;i{:=}0;\mathbf{While}\ i < n\ \mathbf{do}\ (r{:=}r+i;i{:=}i+1)$$
$$C' \equiv r{:=}0;i{:=}0;\mathbf{While}\ i < n\ \mathbf{do}\ (r{:=}r+i;i{:=}i+1;r{:=}r+i;i{:=}i+1).$$

The equivalence between the C and its unrolling C' for even n may be formulated as the relational specification $C \sim C' : T_N \Longrightarrow S_N$ for any $N \geq 0$ and

$$T_N \equiv \{((s,h),(s',h')).\ [\![n]\!]_s = [\![n]\!]_{s'} = 2N\}$$
$$S_N \equiv \{((s,h),(s',h')).\ [\![n]\!]_s = [\![n]\!]_{s'} = [\![i]\!]_s = [\![i]\!]_{s'} = 2N \wedge [\![r]\!]_s = [\![r]\!]_{s'} = 2N^2 - N\}.$$

A proof for this specification using the rule for independent loops instantiates R to

$$T_N \cap \left\{ ((s,h),(s',h')) \;\middle|\; \begin{array}{r} \exists\ I\ I'\ k.\ [\![i]\!]_s = I \wedge [\![i]\!]_{s'} = I' \wedge 0 \leq I \leq 2N \wedge 0 \leq I' \leq 2N \\ \wedge\ 2[\![r]\!]_s = I(I-1) \wedge 2[\![r]\!]_{s'} = I'(I'-1) \wedge I' = 2k \end{array} \right\}$$

where each conjunct applies to either the primed or the non-primed state.

Alternatively, the same specification may be proven using the rule for partially synchronized loops, using the instantiation $W = \emptyset$,

$$R \equiv T_N \cap \left\{ ((s,h),(s',h')) \;\middle|\; \begin{array}{r} \exists I\,I'.\ [\![i]\!]_s = I \wedge [\![i]\!]_{s'} = I' \wedge 0 \leq I, I' \leq 2N \\ \wedge\ 2[\![r]\!]_s = I(I-1) \wedge 2[\![r]\!]_{s'} = I'(I'-1) \\ \wedge\ ((I < N \wedge I' = 2I) \vee (N \leq I \wedge I' = 2N)) \end{array} \right\}$$
$$U \equiv R \cap \{((s,h),(s',h')).\ [\![i]\!]_s < 2N \wedge [\![i]\!]_{s'} < 2N\}$$
$$V \equiv R \cap \{((s,h),(s',h')).\ N \leq [\![i]\!]_s < 2N\},$$

based on the intuition that the first N iterations proceed synchronously, followed by N additional unilateral iterations of the left loop. The entanglement surfaces in the disjunctive final clause in the definition of R.

The above specifications universally quantify over the meta-variable N at Isabelle-level. Using the parametrization mechanism below, we have also performed verifications where N is part of the specification, and shared between pre- and postconditions.

4 Extensions and Applications

We briefly sketch some extensions of our formal framework, and motivating applications. Details of the development are available in [14].

Parametrized simulations. Often, simulations are of interest where the pre- and post-relations employ auxiliary state. We model this situation by endowing the relations with additional arguments, similar to Kleymann's [26] treatment for unary logics.

Definition 2. *For transition systems $\mathcal{T}$ and $\mathcal{T}'$ as before, type $\mathcal{Z}$ of auxiliary states, and parametrized relations $R : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$, we write $\models_{\mathcal{T}}^{\mathcal{T}'} P \sim P' : R \Longrightarrow_{\mathcal{Z}} S$ if for all z, s, s', t, and t' with $(s, P, t) \in \mathcal{T}$ and $(s', P', t') : \mathcal{T}'$, $sR_z s'$ implies $tS_z t'$, where R_z denotes the application of R to parameter z.*

Parametrized simulation can be reduced to nonparametrized simulation using two constructions on transition systems, as follows. The first construction, the product

$$\mathcal{T} \times \mathcal{T}' \equiv \{((s, s'), (P, P'), (t, t')) \mid (s, P, t) \in \mathcal{T} \wedge (s', P', t') \in \mathcal{T}'\}$$

internalizes the two-execution nature of simulations. Second, we define the identity transition system for parameters $\mathcal{Z}$, denoted by $\mathcal{I}_{\mathcal{Z}}$, by $\{(z, *, z) \mid z \in \mathcal{Z}\}$ where $*$ is the unique value of some singleton set of labels.

The following lemma justifies these constructions by relating $\mathcal{Z}$-parametrized behaviour over $\mathcal{T} \times \mathcal{T}'$ to nonparametrized behaviour over $\mathcal{T} \times (\mathcal{T}' \times \mathcal{I}_{\mathcal{Z}})$, where $\overrightarrow{R}$ denotes the relation $\{(s, (s', z)).\ (s, s') \in R\, z\}$ for any $R : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$.

Lemma 6. *For $R, S : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$ we have $\models_{\mathcal{T}}^{\mathcal{T}'} P \sim P' : R \Longrightarrow_{\mathcal{Z}} S$ exactly iff $\models_{\mathcal{T}}^{(\mathcal{T}' \times \mathcal{I}_{\mathcal{Z}})} P \sim (P', *) : \overrightarrow{R} \Longrightarrow \overrightarrow{S}$.*

Instantiating the parametrization mechanism to our language with objects, we may derive proof rules for judgements $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ formally defined as

$$\exists\, \phi.\, \rhd C : Dec_{\mathsf{L}}\, \overrightarrow{R}\, \phi \wedge \rhd C' : Dec_{\mathsf{R}}\, (\overrightarrow{S})^{\sharp}\, \phi^{\sharp}.$$

Here, the operation $\psi^{\sharp} \equiv \{((x, z), x') \mid (x, (x', z)) \in \psi\}$ shifts the auxiliary value z to the left component, so that it is not affected by the execution of C'. By construction, $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ implies $\models_{\mathcal{T}_{\mathsf{Obj}}}^{\mathcal{T}_{\mathsf{Obj}}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$. The proof rules for the system $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ are essentially the same as in Section 3, and are derived by incorporating the operators $(.)^{\sharp}$ and $\overrightarrow{(.)}$ into the construction of witnesses.

Noninterference for objects. A typical use case for the parametrization mechanism consists of noninterference. Following Banerjee-Naumann [7], we consider a notion of indistinguishability that prevents an attacker from observing the precise location chosen during an allocation, and also allow each execution to allocate objects that have no counterpart in the opposite execution. Formally, this is modeled by parametrizing the relation $\sim$ by partial bijections over locations, i.e. sets $\beta \subseteq \mathcal{L}^2$ satisfying $(\ell = \ell_1) \Leftrightarrow (\ell' = \ell_1')$ for any $(\ell, \ell') \in \beta$ and $(\ell_1, \ell_1') \in \beta$.

Naturally, the bijections evolve throughout program execution according to the allocation of fresh objects, but in a conservative manner: the partial bijection relating the final states should be an extension of the one relating the initial states. We therefore parametrize the simulations by bijections, communicating the initial bijection to the postrelation. Indeed, for

$$R_{\mathsf{NI}} = \lambda\,\beta.\ \{(\sigma, \sigma').\ \sigma \sim_\beta \sigma'\} \qquad S_{\mathsf{NI}} = \lambda\,\beta.\ \{(\sigma, \sigma').\ \exists\,\gamma \supseteq \beta.\ \sigma \sim_\gamma \sigma'\}$$

noninterference coincides with $\models^{\mathcal{T}_{\mathsf{Obj}}}_{\mathcal{T}_{\mathsf{Obj}}} C \sim C : R_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} S_{\mathsf{NI}}$ and, in fact, also with $\models^{\mathcal{T}_{\mathsf{Obj}}}_{\mathcal{T}_{\mathsf{Obj}}} C \sim C : S_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} S_{\mathsf{NI}}$. This motivates the definition of the derived forms

$$\begin{aligned} \mathit{LOW}(C) &\equiv \vdash_{\mathsf{Par}} C \sim C : S_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} S_{\mathsf{NI}} \\ \mathit{HIGH}(C) &\equiv \vdash_{\mathsf{Par}} C \sim \mathbf{Skip} : R_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} R_{\mathsf{NI}}, \end{aligned}$$

that interpret, respectively, the judgements for noninterferent and publically unobservable code fragments. As the semantic interpretations S_{NI} and R_{NI} are transparent, the derived rules can be combined with direct uses of the underlying rules for $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ to integrate type-based with logical reasoning.

Error behaviour and separation logic. We have also derived proof rules of unary and relational separation logic, including the appropriate frame rules. The derivations make crucial use of the parametrization mechanism, by instantiating $\mathcal{Z}$ to the type of (relational) assertions. This allows frame assertions to be joined onto the pre- and postconditions in a style reminiscent of Birkedal et al.'s Kripke resource extension [16]. Our encoding is derived from a variant of $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ for a language where null dereferences and attempts to access undefined fields result in a fault/error state. The faultiness of states is exposed in the specifications of the unary and derived relational logics, enabling the interpretation of separation logic judgements to specify equi-fault-avoidance of the two phrases. We include the Isabelle-files of this development in [14] but are prevented from a detailed exposition by page limitations.

5 Discussion

Relational decomposition is a technique for integrating relational logics into stacks of unary verification frameworks [6,4]. We established soundness and completeness of decomposition for general simulations, introduced relational variants of predicate transformers, and studied their relationship to unary transformers. We applied our findings to derive relational program logics, and sketched applications to noninterference and separation logics. Our development is backed up by a formalization in Isabelle/HOL.

The formulation across different transition systems was crucial for our derivation of parametrized simulations. Future work will seek to exploit this flexibility for the verification of refinement and compiler correctness. Work on a relational logic for a bytecode-like language is under way, with a system for formally relating the two language levels as an intended subsequent step. Later, one might aim to support features such as arrays, exceptions, and methods. Our treatment of noninterference in [15] already supports parameterless but possibly recursive procedures, but transferring this development to virtual methods and non-lockstep method invocations is future work.

Concrete relational verification might benefit from formulating relational decomposition more algorithmically, so that the traversal of a program pair emits unary verification tasks, along the line of Terauchi and Aiken's work. Hints for the discovery of relational invariants may potentially arise from Amtoft et al.'s preconditions for conditional information flow [2], Barthe et al.'s product programs [10], from Rhodium's transformation rules [27], or from Tate et al.'s program equivalence graphs [36]. It would also be interesting to compare the expressiveness and usability of our rules for dissonant loops with the rules from translation validation [20], and to investigate how the latter can be justified in a more semantics-oriented fashion.

Natural extensions of noninterference include extensional notions of declassification [8], conditional information flow [3], and the explicit integration of noninterference and separation disciplines, following the work of Amtoft et al. [1]. Magill et al.'s two-step abstractions for reasoning about data structures may provide orientation how ghost variables and program instrumentation interact with separation aspects [29].

A more abstract treatment of our operators can be obtained using relational algebra. As pointed out by a referee, uncurrying $\mathit{Dec}_{\mathsf{L}}\ R\ \phi$ yields $(R\backslash\phi)^{-1}$ while uncurrying $\mathit{Dec}_{\mathsf{R}}\ S\ \phi$ yields the *weakest postspecification* S/ϕ given by $\overline{\phi^{-1};\overline{S}}$. Extending the work of [22,23], Gardiner [19] explores connections between these operators and predicate transformers to study a variation of bisimulation called power simulation. In contrast to our work, predicates and relations are formulated over a single universe.

Barthe et al.'s article [11] includes a self-composed treatment of separation, but restricted to a (termination- and) error-insensitive case and without a fine-grained object control via partial bijections. Reducing error-avoidance of self-composed programs to *equi*-error-avoidance of C and C' appears difficult as the execution of C' is conditional on the nonfaultiness of C's final state.

Saabas and Uustalu show how type derivations yield semantics-preserving proof transformations between pairs of judgements of unary Hoare logics [34].

A long-term goal is the integration of our techniques into verification infrastructures for mainstream languages such as the Verified Software Toolchain for C [5]. As a stepping stone towards this goal, fragments of C such as Spark/Ada [9] may represent a realistic testbed that is both industrially relevant and formally tractable.

Acknowledgments. Andrew Appel encouraged me to revisit the earlier article with Martin Hofmann. The PL group at Princeton and the SAnToS group at Kansas State University provided numerous comments. I am grateful to Dave Naumann and the anonymous referees for their detailed feedback and several additional pointers to the literature.

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: Morrisett and Jones [30], pp. 91–102
2. Amtoft, T., Hatcliff, J., Rodríguez, E.: Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 43–63. Springer, Heidelberg (2010)
3. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.A.: Specification and Checking of Software Contracts for Conditional Information Flow. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Heidelberg (2008)
4. Appel, A.W.: Foundational high-level static analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, Springer, Heidelberg (2008)
5. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011)
6. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resources. Theoretical Computer Science 389(3), 411–445 (2007)
7. Banerjee, A., Naumann, D.: Stack-based access control for secure information flow. Journal of Functional Programming 15, 131–177 (2005)
8. Banerjee, A., Naumann, D.A., Rosenberg, S.: Towards a logical account of declassification. In: Hicks, M.W. (ed.) Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS 2007), pp. 61–66. ACM Press, New York (2007)
9. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley, Reading (2006)
10. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs (2011), `http://software.imdea.org/~ckunz/rellog/long-rellog.pdf`
11. Barthe, G., D'Argenio, P., Rezk, T.: Secure Information Flow by Self-Composition. In: Foccardi, R. (ed.) Computer Security Foundations Workshop, pp. 100–114. IEEE Press, Los Alamitos (2004)
12. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: Fähndrich, M. (ed.) Types in Language Design and Implementation, pp. 103–112. ACM Press, New York (2005)
13. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL 2004), pp. 14–25. ACM Press, New York (2004)
14. Beringer, L.: Relational decomposition – Isabelle/HOL sources (2011), `www.cs.princeton.edu/~eberinge/RelDecompITP2011.tar.gz`
15. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: IEEE Computer Security Foundations Symposium, pp. 233–248. IEEE Press, Los Alamitos (2007)
16. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules. In: Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005), pp. 260–269. IEEE Press, Los Alamitos (2005)
17. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Journal of Symbolic Logic 22(3), 269–285 (1957)
18. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Gorrieri, R. (ed.) Workshop on Issues in the Theory of Security. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS (2003)
19. Gardiner, P.: Power simulation and its relation to traces and failures refinement. Theoretical Computer Science 309(1-3), 157–176 (2003)
20. Goldberg, B., Zuck, L.D., Barrett, C.W.: Into the loops: Practical issues in translation validation for optimizing compilers. Electronic Notes in Theoretical Computer Science 132(1), 53–71 (2005)

21. Hermenegildo, M.V., Palsberg, J. (eds.): Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL 2010). ACM Press, New York (2010)
22. Hoare, C.A.R., He, J.: The weakest prespecification. Information Processing Letters 24(2), 127–132 (1987)
23. Hoare, C.A.R., He, J., Sanders, J.W.: Prespecification in data refinement. Information Processing Letters 25(2), 71–76 (1987)
24. Hunt, S., Sands, D.: On flow-sensitive security types. In: Morrisett, Jones (eds.) [30], pp. 79–90
25. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice-Hall International, Englewood Cliffs (1990)
26. Kleymann, T.: Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs. PhD thesis, LFCS, University of Edinburgh (1998)
27. Lerner, S., Millstein, T.D., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL 2005), pp. 364–377. ACM Press, New York (2005)
28. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
29. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic numeric abstractions for heap-manipulating programs. In: Hermenegildo, Palsberg [21], pp. 211–222
30. Morrisett, J.G., Jones, S.L.P. (eds.): Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL 2006). ACM Press, New York (2006)
31. Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006)
32. Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Not. 35(5), 83–94 (2000)
33. Nipkow, T.: Hoare logics for recursive procedures and unbounded nondeterminism. In: Bradfield, J. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 103–119. Springer, Heidelberg (2002)
34. Saabas, A., Uustalu, T.: Program and proof optimizations with type systems. Journal of Logic and Algebraic Programming 77(1-2), 131–154 (2008)
35. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communication 21, 5–19 (2003)
36. Tate, R., Stepp, M., Lerner, S.: Generating compiler optimizations from proofs. In: Hermenegildo, Palsberg [21], pp. 389–402
37. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
38. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. Journal of Computer Security 4(3), 167–187 (1996)
39. Voronkov, A., Narasamdya, I.: Inter-program properties. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 343–359. Springer, Heidelberg (2009)
40. Yang, H.: Relational separation logic. Theoretical Computer Science 375(1-3), 308–334 (2007)

Structural Analysis of Narratives with the Coq Proof Assistant

Anne-Gwenn Bosser[1], Pierre Courtieu[2], Julien Forest[3], and Marc Cavazza[1]

[1] University of Teesside, School of Computing
http://ive.scm.tees.ac.uk/
[2] Conservatoire National des Arts et Métiers, Laboratoire CEDRIC, Equipe CPR
http://cedric.cnam.fr/
[3] École nationale supérieure d'informatique pour l'industrie et l'entreprise,
Laboratoire CEDRIC, Equipe CPR
http://cedric.cnam.fr/

Abstract. This paper proposes a novel application of Interactive Proof Assistants for studying the formal properties of Narratives, building on recent work demonstrating the suitability of Intuitionistic Linear Logic as a conceptual model. More specifically, we describe a method for modelling narrative resources and actions, together with constraints on the story endings in the form of an ILL sequent. We describe how well-formed narratives can be interpreted from cut-free proof trees of the sequent obtained using Coq. We finally describe how to reason about narratives at the structural level using Coq: by allowing one to prove 2nd order properties on the set of all the proofs generated by a sequent, Coq assists the verification of structural narrative properties traversing all possible variants of a given plot.

Keywords: Applications of Theorem Provers, Linear Logic, Formal Models of Narratives.

1 Introduction

The formalisation of narratives has attracted interest from researchers from many disciplines, not solely for their role as knowledge structures [24], but also for the challenges that their structural properties pose to logical representations [15, 17]. Narratives extend the logic of actions to provide a framework in which causal, temporal and resource consumption aspects are intertwined. Whilst the logical formalisation of actions has become a standard topic in philosophical logic and formal semantics, comparatively little work has addressed the structure of narratives. Initial hopes of developing computational narratology on the same basis as computational linguistics using narrative models developed in the field of humanities [2, 13] have failed due to narratology's formalisms being mostly content ontologies rather than logical or computational formalisms [3].

Addressing this problem from a new perspective, we have recently described in [1] how Linear Logic (LL) [10], and in particular Intuitionistic Linear Logic

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 55–70, 2011.
Springer-Verlag Berlin Heidelberg 2011

(ILL) [11] can provide a suitable conceptual model for Narratives on a structural basis. Narratives are modelled as proofs of a sequent written in Linear Logic which describes initial resources and possible narrative actions. This allows one to naturally express key properties for Narratives (generation of a variety of stories, variability in an open-world assumption, and narrative drive with regards to goals and actions execution) while supporting a return to first principles of narrative action representation (causality and competition for resources). This was not merely an attempt at logically encoding a given narrative: on the contrary the logical formulation supports the description of possible variants of the narrative, including alternative endings, which would be logically consistent. In other words, the ILL formalisation captures the essence of the narrative logic, not simply the accidentality of a given story. Since the manual exploration of proofs to discover story variants can be both tedious and error prone, we decided to support this exploration with proof assistants.

Expanding these early results, we propose here a first step towards the automation of the structural analysis of narratives using the Coq Proof Assistant. We describe how to specify narratives on a structural basis only (causality and resource consumption) in the form of an ILL sequent, and a dedicated ILL encoding into Coq[1], with tactics allowing the discovery of proofs of such a sequent. We also describe how such proofs are interpreted as well-formed narratives. Our encoding of ILL into Coq supports, as has previous work, the assisted generation of ILL proofs, but also assists reasoning about the properties of proofs and on all the possible proofs of an ILL sequent. This allows us to explore second order structural properties, traversing all the narratives which can be generated from a description of initial and atomic narrative actions and resources.

2 Related Works

2.1 Logical Approaches to Narratology

While most of the research in computational narratology has developed empirically, there have been a few logical and formal approaches to narratology, some of which are reviewed here.

A formal grammar for describing narratives has been proposed by [17], supporting the implementation of a system generating linear narratives and relying on temporal logic. While such generated narratives are not able to support an open-world assumption or to take into account the point of view of more than one protagonist, the approach shares with ours the emphasis on narrative causality description which is here embedded in the heart of the formalism.

Grasbon and Braun [12] have used standard logic programming to support the generation of narratives. However their system still relied on a narrative ontology (inspired from Vladimir Propp's narrative functions [23]), rather than on logical properties as first principles. Logic Programming has also been used in [26] for

[1] Source available:
`http://cedric.cnam.fr/~courtiep/downloads/ill_narrative_coq.tgz`

the generation of logically consistent stories. This character-based approach relies on argumentation models developed for autonomous agent systems for resolving the conflicts experienced by protagonists. Our more generic approach relies only on the description of narratives on the structural fundamentals which are action representation and competition for resources.

The concept of narrative action and its impact on the narrative environment is generally considered by narrative theories as the fundamental building block. Therefore AI formalisms dedicated to action semantic representation have been used previously for narrative action description, such as the situation calculus [21]. Linear Logic provides a very elegant solution to the frame problem by allowing the description of narrative actions using an action-reaction paradigm, avoiding the need to specify additional frame axioms for representing actions' non-effects.

The only previous use of LL in a closely related application has first been reported by [4], where the multiplicative fragment of LL is used for scenario validation. Their approach aims at a priori game/scenario design validation, through compilation into Petri Nets, with an emphasis on evidencing reachable states and dead-ends. While providing a relatively friendly computational model, such a fragment is not expressive enough for our purpose.

2.2 Related Applications of Linear Logic

Recent research in computational models of narratives has converged on the use of planning systems: typically, a planner is used to generate a sequence of actions which will constitute the backbone of the narrative [27]. On the other hand, Linear Logic has typically been used for action representation, and Masseron *et al.* [19, 20] have established how LL formalisation could support planning and how the fundamental properties of LL allows a proof in LL to be equated to a plan. While the Intuitionistic fragment of Linear Logic is undecidable, Dixon *et al.* [8,9] use proof-assistant technologies to build and validate plans for dialogues between agents in a Multi-Agents System. The approach we propose here goes, however, beyond the generation of a course of actions as we are interested in studying and verifying second order structural properties, transcending all the narratives which can be generated from a given specification relying on ILL.

While the computational properties of the fragment of Linear Logic we consider are an obstacle for the automation or semi-automation of proof-search (see [18] for a survey of decidability and complexity results for various fragments), the subset-language we use provides some restrictions and additional properties. This is similar to previous use of LL in the field of computational linguistics: [14] identifies usage patterns of LL in meaning assembly analysis [7] ensuring better complexity results than the full considered fragment.

2.3 Proof Assistants Support for LL

Previous work has proposed various encodings of fragments and variants of Linear Logic for proof assistants. In [22], the authors present a shallow embedding of ILL in Coq and perform some simple generic proofs using induction. In [25], the authors present a shallow embedding of the sequent calculus of classical modal linear logic and perform some *epistemic* proofs. In [16] an efficient and easy to use implementation of ILL rules in the Isabelle framework is presented. However our development focuses on properties of proofs (interpreted as narratives) themselves, not just on provability of sequents. As in these previous works we provide some (limited) automation for proving *closed sequents*, but we also provide reasoning lemmas and tactics for reasoning on properties of proofs and even on *all possible proofs of a sequent.*

More recently, Dixon *et al.* [8] have proposed a formalisation of ILL in Isabelle focusing on the generation of verified *plans.* This is certainly the approach that is closest to ours, as it allows reasoning on plans themselves. A notable difference, due to the use of Isabelle, is that plans appear explicitly in the judgments as "extracted proof terms". We do not need this artefact in our formalisation: narratives are pure ILL proof-terms. The relation between the shape of a proof and the properties of the corresponding narrative is, to our knowledge an original use of the proof-as-term paradigm.

3 ILL as a Representational Theory for Narratology

Our approach is based on a formal specification of narrative resources (including narrative actions), initial conditions, and possible ending states in the form of an ILL sequent. We then interpret a given proof of such a sequent as a narrative taking place in an open-world assumption. A sequent may have multiple proofs. It may therefore specify multiple narratives sharing the same initial resources and narrative actions. When interpreting the proof as a narrative, we look for a trace of the use of the $\multimap$ left rule. This rule is interpreted as the execution of a narrative action. Other rules have an interpretation reflecting the structure of the narrative, such as an external branching choice in an open-world assumption (for instance, end-user interaction), or a concurrency relationship between different subsets of the narrative with independent resource requirements.

3.1 Modelling of Narratives Specification through an ILL Sequent

The subset language of ILL we use for this paper allows the description of the initial resources of the narrative, the available narrative actions, and constraints on the possible ending states of the narrative. Key to our interpretation, narrative actions are modelled using $\multimap$ which allows a precise description of their impact on the narrative environment. As we work in an open world assumption, external impact on the narrative (for instance user interaction) is modelled by using the $\oplus$ connector for describing choices between possible narrative actions, and by using & for describing a choice between two possible ending states.

Such a specification of narratives encompasses the description of the available resources and states of the narratives, the description of the semantics of narrative actions through their impact on the context of execution, and the possible ending states of the narrative. The *initial* sequent, which models this specification, thus takes the form $\mathcal{R}, \mathcal{A} \vdash \texttt{Goal}$, where $\mathcal{R}$ is a multiset representing resources and initial conditions, $\mathcal{A}$ is a multiset representing the possible narrative actions, and *Goal* a formula representing the possible ending state of the narrative. A sequent thus provides the knowledge representation base of a set of narratives.

We refer the reader to [11] for a description of ILL sequent calculus and to [1] for a more detailed description of the use of ILL operators for our purpose, which served as a base for the subset narrative specification language defined in this paper (Figure 1). These restrictions on the structure of the initial sequent will enforce properties on the possible proofs, which can be verified using Coq.

We use here an extract of Flaubert's classical *Madame Bovary* novel[2] as a running example: facing public humiliation, Emma is unsucessfully looking for the help of Guillaumin, Binet and Rodolphe, before ingesting the poison she has previously located. We start from an identification of atomic resources and simple narrative actions: we add alternatives to some of the narrative actions occurring in the novel, inspired by each of the character's possible choices and introduce the possibility of two different endings (in one of those Emma survives). Based on this identification, we model narrative context and goals as an ILL sequent.

$$
\begin{array}{ll}
\texttt{Res} & ::= \texttt{1} \mid \texttt{atom} \mid \texttt{Res} \,\&\, \texttt{Res} \mid \texttt{Res} \otimes \texttt{Res} \mid !\texttt{Res} \\
\texttt{Act} & ::= \texttt{1} \mid \texttt{CRes} \multimap \texttt{Context} \mid \texttt{Act} \oplus \texttt{Act} \mid \texttt{Act} \,\&\, \texttt{Act} \mid !\texttt{Act} \\
\texttt{Goal} & ::= \texttt{1} \mid \texttt{atom} \mid \texttt{Goal} \otimes \texttt{Goal} \mid \texttt{Goal} \oplus \texttt{Goal} \mid \texttt{Goal} \,\&\, \texttt{Goal} \\
\texttt{CRes} & ::= \texttt{1} \mid \texttt{atom} \mid \texttt{CRes} \otimes \texttt{CRes} \\
\texttt{Context} & ::= \texttt{Res} \mid \texttt{Act} \mid \texttt{Context} \otimes \texttt{Context}
\end{array}
$$

Fig. 1. Syntactic categories for narrative sequents

Resources of a Narrative. $\texttt{Res}$ specifies the syntactic category for $\mathcal{R}$. The formula $\texttt{Res}_1 \,\&\, \texttt{Res}_2$, expresses the availability of one of the resources. One only of $\texttt{Res}_\texttt{i}$ will be used and the choice depends on the proof found, and can vary depending on the branches of the proof. This allows us to describe how the initial conditions can adapt to a given unfolding of the story. The formula $\texttt{Res}_1 \otimes \texttt{Res}_2$ allows one to express the availability of both resources. The formula $!\texttt{Res}$ allows one to express the unbounded availability of the resource $\texttt{Res}$. The atomic resources in our example are P for poison, R for a discussion with Rodolphe, B for a discussion with Binet, and G for a discussion with Guillaumin. We chose to not enforce the consumption of the resources P and B in our example, and therefore model $\mathcal{R} = P \,\&\, 1, R, G, B \,\&\, 1$.

Narrative Actions Representation. $\texttt{Act}$ specifies the syntactic category for$\mathcal{A}$. A simple narrative action is of the form $\texttt{CRes} \multimap \texttt{Context}$, where $\texttt{CRes}$ is a finite resource description and $\texttt{Context}$ a multiplicative conjunction of resources and actions. Its semantics is thus precisely defined in terms of how it

[2] The plot of Madame Bovary is one naturally inclined to contain key decisions and prone to suggest what-if analyses.

affects the execution environment: to the execution of a narrative action in the narrative corresponds the application of the $\multimap$ left rule in the proof, consuming the finite amount of resources modelled by `CRes` (in the subset-language we use for this paper, actions only consume resources) and introducing in the sequent context the formula `Context` which models resources and actions made available by this execution.

For our example, we model the following simple narrative actions:

$\mathrm{S}\multimap\mathrm{A}$	Emma sells herself which saves her life.
$\mathrm{E}\multimap\mathrm{A}$	Emma escapes with Rodolphe (which saves her life).
$\mathrm{P}\multimap\mathrm{D}$	Emma ingests poison and dies.
$\mathrm{R}\multimap 1$	Emma has a conversation with Rodolphe. This does not alter her situation (non productive).
$\mathrm{R}\multimap\mathrm{E}$	Emma talks to Rodolphe. They agree to escape together.
$\mathrm{G}\multimap 1$	Emma has a conversation with Guillaumin. This does not alter her situation (non productive).
$\mathrm{G}\multimap\mathrm{S}$	Emma discusses her situation with Guillaumin. As a result, Emma accepts to sell herself in exchange for Guillaumin's help.
$\mathrm{B}\multimap 1$	Emma has a conversation with Binet. This does not alter her situation (non productive).
$\mathrm{B}\multimap\mathrm{S}$	Emma discusses her situation with Binet. As a result, Emma accepts to sell herself in exchange for Binet's help.

Narrative actions can be composed. In particular, they can be combined for offering two types of choices. A composed action $\mathtt{Act}_1 \oplus \mathtt{Act}_2$ corresponds to a choice between two possible actions, with both possibilities leading to well-formed alternative narratives. This is used for modelling the impact of events external to the narrative in an open-world assumption (for instance, user interaction). When such a formula is decomposed using the $\oplus$ rule, the two sub-proofs, which require proving the sequent with each of the sub-formula replacing the composed action, are interpreted as the two possible unfoldings of the story. The proof thus ensures that each possible subsequent narrative is well-formed. A composed action $\mathtt{Act}_1 \,\&\, \mathtt{Act}_2$ corresponds to a choice depending on the proof found. If both choices successfully produce a different proof of the sequent, this will be interpreted as two different narratives.

In our example, we model:
$\mathcal{A} = !(S \multimap A), (E \multimap A) \,\&\, 1, (P \multimap D) \,\&\, 1, (R \multimap 1) \,\&\, (R \multimap E), (G \multimap 1) \oplus (G \multimap S), 1 \oplus ((B \multimap S) \,\&\, (B \multimap 1))$

The composed action $(G \multimap 1) \oplus (G \multimap S)$ reflects branching narrative possibilities depending on the impact of external events in an open-world assumption, while $(E \multimap A) \,\&\, 1$ can potentially generate a narrative where the narrative action corresponding to $E \multimap A$ occurs, or not.

Narrative Ending States. `Goal` specifies the syntactic category for ending state. $\mathtt{Goal}_1 \otimes \mathtt{Goal}_2$ expresses that both $\mathtt{Goal}_\mathtt{i}$ states are accessible at the end of the narrative. $\mathtt{Goal}_1 \oplus \mathtt{Goal}_2$ expresses that either $\mathtt{Goal}_\mathtt{i}$ state is accessible, and the choice depends on the proof found and might differ depending on the unfolding branch of the narrative. $\texttt{Goal}_1 \,\&\, \texttt{Goal}_2$ expresses that either state should be accessible, and this choice depends on an event external to the narrative, such as user interaction for instance.

In our example, we model that a given narrative could possibly provide two different endings: from the atomic goals A (for Emma is alive) and D (for Emma is dead), we specify the right-hand side formula $A \oplus D$. This concludes the specification of our example of narrative into an ILL sequent.

Stability of the Representation. Given an ILL sequent respecting the grammar described in Figure 1, all the sequents appearing in the proof will be of the form $\Gamma \vdash G$, where $\forall F \in \Gamma$, F is a `Context` formula and G is a `Goal` formula. In other words all the sequents appearing in such a proof will be composed of a context describing resources and actions of a narrative, and of a right-hand side formula representing constraints on the ending state of the narrative.

More formally, we define the following properties on sequents and proofs:

Definition 1. *Let s be a sequent of the form $\Gamma \vdash G$, we say that s is* well formed *if $G \in$ **Goal** and $\forall F \in \Gamma, f \in$ **Context**. We shall write* WF(s).

Definition 2. *A property P on sequents is said to* hold for a proof h *of a sequent s if it holds for all sequents of the proof h above s. We shall note* WF(h).

Lemma 1. WF *is stable for ILL, that is for any sequent s such that* WF(s) *and any proof h of s,* WF(h).

The proof of this property in Coq is described later in section 4.4.

3.2 Interpreting a Proof as a Narrative

Narratives are interpreted from proofs, from a structured trace of execution of the $\multimap$ left rule. Other ILL rules of particular significance for this interpretation are the $\oplus$ left, and $\otimes$ and $\&$ right rules (we refer the reader to [11] for a description of ILL sequent calculus). Narratives are obtained from proofs using the ν function described in Figure 2. Narrative are thus described using simple narrative actions (modelled in the initial sequent using the $\multimap$ connector), and the following list of operators:

- $\succ$ is a precedence relationship, defining a partial order on the execution of narrative actions: $\nu = \nu_1 \succ \nu_{action} \succ \nu_2$ is a narrative where the narrative ν_1 precedes the narrative action ν_{action} which precedes the narrative ν_2.
- $\triangledown$ is a branching of the narrative in an open-world assumption: $\nu = \nu_1 \triangledown \nu_2$ is a narrative where both sub-narratives ν_1 and ν_2 are possible, but only one will actually be unfolded, depending on an external event in an open-world assumption (such as user interaction for instance).
- $||$ represents a concurrency relationship between two narratives: $\nu = \nu_1 || \nu_2$ is a narrative consisting of both ν_1 and ν_2 where the two sub-narratives will be unfolded concurrently and independently.

$$\frac{\Gamma \vdash A : \nu_1 \qquad \Delta, B \vdash C : \nu_2}{\Gamma, \Delta, A \multimap B \vdash C : \nu_1 \succ \nu_{A \multimap B} \succ \nu_2}\ (\multimap_{left}) \qquad \frac{}{\Gamma \vdash A : \emptyset}\ (Leaf\ rules)$$

$$\frac{\Gamma \vdash A : \nu_1 \qquad \Delta \vdash B : \nu_2}{\Gamma, \Delta \vdash A \otimes B : \nu_1 \| \nu_2}\ (\otimes_{right}) \qquad \frac{\Gamma \vdash A : \nu}{\Gamma' \vdash A' : \nu}\ (Unary\ rules)$$

$$\frac{\Gamma, A \vdash C : \nu_1 \qquad \Gamma, B \vdash C : \nu_2}{\Gamma, A \oplus B \vdash C : \nu_1 \nabla \nu_2}\ (\oplus_{left}) \qquad \frac{\Gamma \vdash A : \nu_1 \qquad \Gamma \vdash B : \nu_2}{\Gamma \vdash A \,\&\, B : \nu_1 \nabla \nu_2}\ (\&_{right})$$

Fig. 2. Proof to Narrative Interpretation Function ν: the function is defined recursively on sub-proofs from the last applied ILL rule. $\nu_{A \multimap B}$ is the narrative action initially specified using the formula $A \multimap B$.

Using Coq with simple tactics, we can generate a proof of the sequent $\mathcal{R}, \mathcal{A} \vdash A \oplus D$ specified in Section 3.1. Such a proof can then be interpreted as a given narrative (Figure 3): depending on the impact of an external event (for instance, end-user interaction), the story can first take two different paths. In one of them (right sub-proof tree), the discussion with Guillaumin leads to an offer to help Emma, and to two different paths both ending with Emma's survival. In the second one (left sub-proof tree), the discussion leads to another two paths, one of which ending with Emma's suicide depending on the impact of an external event on the choice of course of action corresponding to the discussion with Rodolphe.

4 Using the Coq Proof Assistant for Narrative Properties Analysis

In this section, we will first describe how, based on our interpretation of proofs as narratives and our ILL encoding into Coq, a proof assistant supports the building of coherent narratives from initial specifications.

This naturally leads one to wonder, given an initial specification, what are the characteristics of a *well-formed* narrative. In order to answer this, we need to be able to express properties regarding the set of all possible proofs of a given sequent, and to formally evidence structural properties which are verified by all the narratives generated by a given specification: we need to be able to express and prove properties by reasoning about proofs and sets of proofs.

We thus discuss in this section how we have been taking advantage of this proof-as-term paradigm in order to verify properties regarding all the proofs corresponding to narrative specifications as defined in Figure 1, and to verify an example of structural property on the set of all the narratives generated by a given sequent specification.

4.1 ILL Encoding into Coq

Formulae, proofs and corresponding convenient (Unicode) notations are defined as follows. Type `env` is an instance of multisets equipped with an (setoid) equality

1. Sequent Description

Initial Resources $\mathcal{R}$	$P \,\&\, 1, R, G, B \,\&\, 1$
Narrative actions $\mathcal{A}$	$!\,(S \multimap A), (E \multimap A) \,\&\, 1, (P \multimap D) \,\&\, 1, (R \multimap 1) \,\&\, (R \multimap E),$ $(G \multimap 1) \oplus (G \multimap S), 1 \oplus ((B \multimap S) \,\&\, (B \multimap 1))$

2. Sketch of the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\multimap_{left}\!: P \multimap D}{\multimap_{left}\!: R \multimap 1}}{\multimap_{left}\!: B \multimap 1} \quad \dfrac{\multimap_{left}\!: E \multimap A}{\multimap_{left}\!: R \multimap E}}{\oplus left : 1 \oplus ((B \multimap S) \,\&\, (B \multimap 1))}}{\multimap_{left}\!: G \multimap 1} \quad \dfrac{\dfrac{\dfrac{\multimap_{left}\!: S \multimap A \quad \dfrac{\multimap_{left}\!: S \multimap A}{\multimap_{left}\!: B \multimap 1}}{\oplus left : 1 \oplus ((B \multimap S) \,\&\, (B \multimap 1))}}{\multimap_{left}\!: R \multimap 1}}{\multimap_{left}\!: G \multimap S}}{\oplus_{left} : (G \multimap 1) \oplus (G \multimap S)}}{\mathcal{R}, \mathcal{A} \vdash A \oplus D}$$

3. Interpreted narrative:
$(\nu_{G \multimap 1} \succ ((\nu_{B \multimap 1} \succ \nu_{R \multimap 1} \succ \nu_{P \multimap D}) \nabla (\nu_{R \multimap E} \succ \nu_{E \multimap A}))) \nabla$
$(\nu_{G \multimap S} \succ \nu_{R \multimap 1} \succ (\nu_{S \multimap A} \nabla (\nu_{B \multimap 1} \succ \nu_{\multimap A}))$

Fig. 3. ILL specification of the end of Emma Bovary

relation == and Vars.t (type of atomic propositions) is implemented[3] as ℕ in the following:

```
      i   formula :  y   :=
| Proposition : Vars.t → formula | Implies : formula → formula → formula
| Otimes : formula → formula → formula | One : formula
| Oplus : formula → formula → formula | Zero : formula | Top : formula
| Bang : formula → formula | And : formula → formula  → formula.

N    i   "A ⊸ B" := (Implies A B).
(* ...Other connectives... *)
N    i    "x :: Γ" := (add x G). (* Environment operation *)
N    i    "x \ Γ" := (remove x G). (* Environment operation *)
N    i    "x ∈ Γ" := (mem x G). (* Environment operation *)

      i   ILL_proof: env → formula → Prop:=
|Id: ∀ Γ p, Γ == {p} → Γ ⊢ p
|Impl_R: ∀ Γ p q,  p::Γ ⊢ q  →  Γ ⊢ p ⊸ q
|Impl_L: ∀ Γ Δ Δ' p q r, (p ⊸ q) ∈ Γ  →  (Γ\(p ⊸ q)) == Δ ∪ Δ'  →  Δ ⊢ p
                                                  →  q:: Δ' ⊢ r  →  Γ ⊢ r
|Times_R: ∀ Γ Δ Δ’ p q, Γ == Δ ∪ Δ’  →  Δ ⊢ p  →  Δ’ ⊢ q  →  Γ ⊢ p ⊗ q
|Times_L: ∀ Γ p q r, (p ⊗ q) ∈ Γ  →  q::p::(Γ\(p ⊗ q)) ⊢ r  →  Γ ⊢ r
|One_R: ∀ Γ, Γ == ∅  →  Γ ⊢ 1
|One_L: ∀ Γ p, 1 ∈ Γ  →  (Γ\1) ⊢ p  →  Γ ⊢ p
|And_R: ∀ Γ p q, Γ ⊢ p  →  Γ ⊢ q  →  Γ ⊢ (p & q)
|And_L₁: ∀ Γ p q r, (p & q) ∈ Γ  →  p::(Γ\(p&q)) ⊢ r  →  Γ ⊢ r
|And_L₂: ∀ Γ p q r, (p & q) ∈ Γ  →  q::(Γ\(p&q)) ⊢ r  →  Γ ⊢ r
```

[3] By functorial application.

```
|Oplus_L: ∀ Γ p q r, (p ⊕ q) ∈Γ → p::(Γ\(p ⊕ q)) ⊢ r → q::(Γ\(p ⊕ q)) ⊢ r
                                                                          → Γ⊢r
|Oplus_R1: ∀ Γ p q, Γ⊢p → Γ⊢p ⊕ q
|Oplus_R2: ∀ Γ p q, Γ⊢q → Γ⊢p ⊕ q
|T_: ∀ Γ, Γ⊢⊤
|Zero_: ∀ Γ p, 0∈Γ → Γ⊢p
|Bang_D: ∀ Γ p q, !p∈Γ → p::(Γ\(!p)) ⊢ q → Γ⊢q
|Bang_C: ∀ Γ p q, !p∈Γ → !p::Γ⊢q → Γ⊢q
|Bang_W: ∀ Γ p q, !p∈Γ → Γ\(!p) ⊢ q → Γ⊢q
where " x⊢y ":= (ILL_proof x y).
```

Notice the use of the form "$\phi \in \Gamma \to \Gamma \vdash \ldots$" instead of "$\phi, \Gamma \vdash \ldots$". This formulation avoids tedious manipulations on the environment to match rules. Simple tactics allow one to apply rules and premisses of the form $\phi \in \Gamma$ are discharged automatically (on closed environments) by reduction. As we are looking for a trace of the execution of the narrative actions through the application of the $\multimap$ left rule, we are only searching for cut-free proofs and thus do not provide the Cut rule.

The Coq command `Program Fixpoint` allows one to define rather easily dependently typed fixpoints and pattern matchings on terms of type `x ⊢ y`. For instance one can define the ν function described in section 3.2 as follows:

```
Program Fi   i   ν Γ φ (h: Γ ⊢ φ) {struct h}: narrative :=
m   h h wi h
|Id _ _ p ⇒ ∅
|Impl_L Γ Δ Δ' p q r _ _ x x0 ⇒ (ν Δ p x) ≻ [Implies p q] ≻ (ν (q::Δ') r x0)
|And_R Γ p q x x0 ⇒ (ν _ _ x) ▽ (ν _ _ x0)
|Times_R Γ Δ Δ' p q heq x x0 ⇒ (ν Δ p x) || (ν Δ' q x0)
  ...
  .
```

4.2 Well-Formed Narrative Generation: Proving an ILL Sequent In Coq

Our encoding of ILL into Coq can be used simply with the aim of generating a *well-formed* story, from a sequent specification. We provide a set of simple tactics assisting the user in unfolding a proof, thus constructing a proof-term which will subsequently be interpreted as a narrative.

As an example, let us consider the sequent given below presented in Figure 3, corresponding to the end of Emma Bovary.

```
Emma: {P&1, R, G, B&1, !(S ⊸ A), (E ⊸ A)&1, (P ⊸ D)&1,
          (R ⊸ 1)&(R ⊸ E), (G ⊸ 1) ⊕ (G ⊸ S), 1⊕(B ⊸ S)&(B ⊸ 1)} ⊢ A ⊕ D.
```

One can, for example, apply the $\oplus_L$ rule to consider the alternative offered by external choice `(G ⊸ 1) ⊕ (G ⊸ S)`. This is achieved by tactic: `oplus_l (G ⊸ 1) (G ⊸ S)` that leaves with two subgoals. The first one is `{G ⊸ 1, P&1, R, G, ... } ⊢ A ⊕ D` and allows for rule $\multimap_{left}$ rule to perform a narrative action consuming `G` using tactic: `impl_l G 1`.

The succesful proof of this sequent unravels the narrative structure by only producing the set of alternative actions consistent with the baseline plot description.

4.3 Stability of an ILL Narrative Sequent: Well-Formed Sequents

As we have briefly mentioned in section 3.1, the subset of ILL (Figure 1) we consider is stable.This is an important property as it allows one to disregard the use of certain ILL rules (for instance $\multimap_{right}$). It can thus simplify the verification of narrative properties.

In order to use this fact, we provide a proof of stability of WF for ILL (property 1). To this end, we prove the stability of WF for each rule as follows: first the grammar of Figure 1 is defined by the following (mutual) inductive properties:

```
    i   Act : formula → Prop := (* Act *)
| A1: Act 1
| A2:∀ φ1 φ2, CRes φ1 → Context φ2 → Act (φ1 ⊸ φ2)
| A3: ∀ φ1 φ2, Act φ1 → Act φ2 → Act (φ1 ⊕ φ2)
| A4: ∀ φ1 φ2, Act φ1 → Act φ2 → Act (φ1 & φ2)
| A5: ∀ φ, Act φ → Act (!φ)
wi h CRes: formula → Prop:= (* CRes *)
| CRes1: CRes 1
| CRes2: ∀ n, CRes (Proposition n)
| CRes3: ∀ φ1 φ2, CRes φ1 → CRes φ2 → CRes (φ1 ⊗ φ2)
wi h Context: formula → Prop:= (* Context *)
| Context1:∀ φ, Act φ → Context φ
| Context2:∀ φ, Res φ → Context φ
| Context3: ∀ φ1 φ2, Context φ1 → Context φ2 → Context (φ1 ⊗ φ2)
wi h Res: formula → Prop:= (* Res *)
  R1: Res 1
| R2: ∀ n, Res (Proposition n)
| R3: ∀ φ, Res φ → Res (!φ)
| R4: ∀ φ1 φ2, Res φ1 → Res φ2 → Res (φ1 ⊗ φ2)
| R5: ∀ φ1 φ2, Res φ1 → Res φ2 → Res (φ1 & φ2).

    i         : formula → Prop :=
| G1:      1
| G2: ∀ n,      (Proposition n)
| G3: ∀ φ1 φ2,      φ1 →      φ2 →      (φ1 ⊗ φ2)
| G4: ∀ φ1 φ2,      φ1 →      φ2 →      (φ1 ⊕ φ2)
| G5: ∀ φ1 φ2,      φ1 →      φ2 →      (φ1 & φ2).

  i i i  WF Γ f (h:Γ⊢f):=      f ∧∀ g:formula, g∈Γ → Context g.
```

Notice that well-formedness (WF) of a sequent ($\Gamma \vdash$ f) is stated as a property of a proof h of such a sequent (see lemma Grammar_Stable below). Then the stability for each rule is given by an inductive property Istable mirroring the type of proofs, stating that a property pred holds for all premisses sequents of all rules.

Istable *e f h* is true when pred holds for all nodes above the root of h (it does not have to hold for the root itself). Notice that *e* and *f* are declared implicit (using {.}) and can therefore sometimes be omitted.

```
    i   Istable: ∀ {e} {f} (h: e ⊢ f) , Prop :=
| IId: ∀ Γ p heq, Istable (Id Γ p heq)
| IImpl_R: ∀ Γ p q h, pred h  →  Istable h  →  Istable (Impl_R Γ p q h)
| IImpl_L: ∀ Γ Δ Δ' p q r hin heq h h', pred h → pred h'
   → Istable h → Istable h' → Istable (Impl_L Γ Δ Δ' p q r hin heq h h')
| ...
```

The stability of the grammar is proved by the following lemma, stating that any proof h of a well-formed sequent is necessarily well-formed itself:

```
  mm  Grammar_Stable: ∀ Γ ϕ (h:Γ ⊢ ϕ), WF h → Istable WF h.
```

It is proved by induction on h.

4.4 Second Order Analysis of Narratives Specification

We consider in this section the reachability of a given ending state regardless of the impact of external events in an open-world assumption, as an example of an interesting structural property. When considering a given proof (and narrative), this property is not difficult to check. We build a (dependently typed) function (check) which decides this property for a closed proof. That is, it returns trueP if at least one branch contains an application of the $\oplus_{R1}$ rule with $\phi_1 \vdash \phi_r$ on the right premise. We then test it on the proof we found of lemma Emma above and the sequent A ⊢ D. boolP stands for Prop-sorted booleans and ?= is the equality decision over formulae.

```
Program Fi   i   check ϕ1 ϕr {e} {f} (h: e ⊢ f) {struct h}: boolP :=
m   h h wi h
  | One_R _ _ ⇒ falseP
  | One_L Γ p _ x ⇒ check ϕ1 ϕr x
  | Oplus_R1 Γ p q x ⇒
     i  andP (p ?= ϕ1) (q ?= ϕr)  h   trueP       check ϕ1 ϕr x ...
    .
Eval vm_compute in check A D Emma. (* true *)
```

What would be much more interesting from a structural analysis point of view would be to prove that this property is valid regardless of the proof of the sequent (check should return trueP for *any proof of* Emma). In our interpretation, this would mean that for every narrative possibly generated by the initial specification, a given end state is reachable. This is much more difficult to prove using Coq as there is a potentially infinite set of such proofs. We tackle this problem using a variety of means. First, we define a notion of equivalence between proofs. Second, we define an incremental method to avoid proving several times the same property. A description of these two techniques follows.

Identify Proofs Corresponding to the Same Tree. We identify proofs that differ only by the way side premisses (like p ∈ Γ or Γ ∈ Δ ∪ Δ') are proven. We then prove that check and other definitions are compatible with this equivalence.

```
      i   eq: ∀ Γ Γ' f, (Γ ⊢ f) → (Γ' ⊢ f) → Prop :=
| EQId: ∀ Γ1 Γ2 f heq heq', eq (Id Γ1 f heq) (Id Γ2 f heq')
| EQImpl_R:∀ Γ1 Γ2 p q h h', eq h h' → eq (Impl_R Γ1 p q h) (Impl_R Γ2
p q h')
| EQTimes_R: ∀ Γ1 Δ1 Δ3 Γ2 Δ2 Δ4 p q heq heq' h1 h3 h2 h4,
  eq h1 h3 → eq h2 h4
  → eq (Times_R Γ1 Δ1 Δ3 p q heq h1 h2) (Times_R Γ2 Δ2 Δ4 p q heq' h3 h4)
| ...
```

```
  mm  eq_compat_check : ∀ f1 f2 Γ Γ' ϕ (h1:Γ⊢ϕ) (h2:Γ'⊢ϕ),
                                         eq h1 h2  →  check f1 f2 h1 = check f1 f2 h2.
```

An interesting consequence is that one can substitute an environment with a provably equal one in our proofs:

```
  mm  eq_env_compat_check : ∀ f1 f2 Γ Γ' ϕ (h1:Γ⊢ϕ) (h2:Γ'⊢ϕ),
                                          eq Γ Γ'  →  check f1 f2 h1 = check f1 f2 h2.
```

This lemma is heavily used in our automated tactics described in the next section.

Property Validation on All the Proofs of a Sequent. We can also prove that some property holds for all proofs of a given closed sequent. We developed a method for this kind of proofs which can be automated using an external tool. This method should work for properties than one can define as a boolean function over ILL proofs (i.e. of type $\forall \Gamma\phi, \Gamma \vdash \phi \rightarrow$ `boolP`). This method involves intricate lemmas and tactics allowing one to explore all possible proofs of a sequent. This amounts in particular to detect unprovable goals as soon as possible. This is made possible in some cases by generic lemmas about the unprovability of a sequent $\Gamma \vdash \phi$. For instance the following (meta) unprovability result is proved and used:

Lemma 2. *If a variable $v \in \Gamma$ does not appear in the left-hand side of a $\multimap$ in any (sub-)formula of Γ and do not appear in ϕ, then $\Gamma \vdash \phi$ has no proof.*

Our tactics detect such patterns in the hypothesis of a goal g and discharge g by absurdity. The proof strategy applies to a goal G of the form: $\forall h\!:\!\Gamma \vdash \phi{,}f\ h =$ `trueP` and proceeds by building a database of previously proved lemmas as described in algorithm 1.

The use of the lemmas database prevents proving the same lemma twice. The application of previous lemmas is eased by the use of `eq_env_compat_check` (described in previous section). Using this method we manage to prove several non-trivial properties, including the reachability property mentioned earlier.

We have modelled the following simple narrative actions, which give another perspective on the end of Madame Bovary:

B⊸S	A discussion with Binet: Emma accepts the idea of selling herself
B⊸R	A discussion with Binet: Emma decides to go and see Rodolphe
G⊸B	A discussion with Guillaumin: Emma decides to go and see Binet
G⊸S	A discussion with Guillaumin: Emma accepts the idea of selling herself

These actions, together with initial resources, can be used to model the following narrative specification: the outcome of the discussion with Binet will be determined by the proof found, while an external event (in an open-world assumption) which decides between the two possible outcomes of the discussion with Guillaumin.

Two possible ending states are specified for this narrative: Emma is ready to sell herself to improve her situation (S) or prepared to have a discussion with Rodolphe. We want to prove that whatever the narrative generated by this specification, there is always a possible sub-narrative in the open-world assumption which allows for the ending state S to be reached. The corresponding sequent modelled in Coq is:

```
$M(\Gamma \vdash \phi)$:
foreach rule $r$ applicable to $\Gamma \vdash \phi$ do
    foreach sequent $\Delta \vdash \psi$ of the premisses of $r$ do
        if f h = trueP then OK;
        else if $\Delta \vdash \psi \in DB$ then apply $DB(\Delta \vdash \psi)$ and OK;
        else if unprovability tactics applies on $\Delta \vdash \psi$ then OK by absurdity;
        else
            prove new lemma $l : \forall h{:}\Delta \vdash \psi, f\ h = $ trueP using $M(\Delta \vdash \psi)$;
            store $l$ in $DB$; apply $l$;
        end
    end
end
```

Algorithm 1: Proof method for properties of the form: $\forall h{:}\Gamma \vdash \phi, f\ h = \mathtt{trueP}$.

$$s = \{\mathtt{G}, ((\mathtt{B} \multimap \mathtt{S}) \& (\mathtt{B} \multimap \mathtt{R})) \& 1, (\mathtt{G} \multimap \mathtt{B}) \oplus (\mathtt{G} \multimap \mathtt{S}) \vdash \mathtt{S} \oplus \mathtt{R}\}$$

We have proved that this sequent is such that ∀ `(h:s), check _ _ h = trueP`. This proof uses 47 auxiliary lemmas, while the sequent only offers a low-level of generativity. Each lemma is proved automatically but currently the lemmas are stated by hand. We discuss briefly how we plan to automate the lemmas generation in the conclusion of this paper.

In order to show that provided adequate automation our technique can scale on sequents offering a high level of generativity, we proved a similar reachability property on the following sequent which uses narrative actions described in Figure 3:

$$s_1 = \{\mathtt{P}\&1, (\mathtt{S} \multimap \mathtt{A})\&1, (\mathtt{E} \multimap \mathtt{A})\&1, (\mathtt{P} \multimap \mathtt{D})\&1, \mathtt{S}\}) \vdash (\mathtt{A} \oplus \mathtt{D})$$

As the sequent is more generative, this proof uses 283 auxiliary lemmas.

5 Conclusion

In this paper, we have shown that the Coq proof assistant is a powerful tool for studying and verifying structural properties of narratives modelled using Intuitionistic Linear Logic.

We have provided a method for encoding narratives specifications into an ILL sequent, encompassing narrative actions and initial resources description, and described an encoding of ILL into Coq which allows one to build well-formed narratives from proofs of such a sequent.

The encoding we have proposed makes use of the proof-as-term paradigm and allows one to verify structural properties of narratives transcending all narratives generated by such a specification. This allows one to study resource-sensitive and causality relationships emerging from the initial specification. From a low-level description of the semantics of narrative actions, we are thus able to obtain high-level semantics properties regarding the narrative.

Now that we have shown that our encoding and our proof method allows for automated heuristics, we plan to implement certifying external procedures in a similar fashion than previous work of authors [5,6]. More precisely we plan to 1) implement a Coq script generator that will generate the lemmas statements and proof following ideas of section 4.4 and 2) prove more unprovability results, like lemma 2, in order to tame a bit more the combinatorial explosion of ILL proofs. The need to prove properties on proofs themselves forces the use of dependently typed programming style, which happens to be uncommon, especially on sort Prop on which elimination is limited. The experience was however successful.

This work therefore opens new perspectives on the design and understanding of computational models of narratives. A particularly interesting avenue to explore concerns the search for normalised forms of narratives, for instance offering the highest possible degree of sub-narratives parallelism relying on resource independence. Such normalisation procedures can rely on dedicated proof-search algorithms, complementing our existing encoding. This work is also a first step towards the assessment of story variance on a structural and formal basis: based on the definition of equivalence relationships between proofs, and further, between their narrative interpretations, we plan to investigate formally what makes stories differ and propose metrics which would allow one to evaluate narrative specifications.

Acknowledgments. This work has been partly funded by the European Comission under grant agreement IRIS (FP7-ICT-231824).

References

1. Bosser, A.G., Cavazza, M., Champagnat, R.: Linear logic for non-linear storytelling. In: ECAI 2010 Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press, Amsterdam (2010)
2. Brémond, C.: Logique du Récit. Seuil (1973)
3. Cavazza, M., Pizzi, D.: Narratology for interactive storytelling: A critical introduction. In: Göbel, S., Malkewitz, R., Iurgel, I. (eds.) TIDSE 2006. LNCS, vol. 4326, pp. 72–83. Springer, Heidelberg (2006)
4. Collé, F., Champagnat, R., Prigent, A.: Scenario analysis based on linear logic. In: ACM SIGCHI Advances in Computer Entertainment Technology (ACE). ACM Press, New York (2005)
5. Contejean, É., Courtieu, P., Forest, J., Paskevich, A., Pons, O., Urbain, X.: A3PAT, an Approach for Certified Automated Termination Proofs. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010), pp. 63–72. ACM, New York (2010)
6. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: The CiME Rewriting Toolbox, Version 3, `http://cime.lri.fr`
7. Dalrymple, M., Lamping, J., Pereira, F.: Linear logic for meaning assembly. In: Proceedings of the Workshop on Computational Logic for Natural Language Processing (1995)

8. Dixon, L., Smaill, A., Bundy, A.: Verified planning by deductive synthesis in intuitionistic linear logic. In: ICAPS Workshop on Verification and Validation of Planning and Scheduling Systems (2009)
9. Dixon, L., Smaill, A., Tsang, T.: Plans, actions and dialogues using linear logic. Journal of Logic, Language and Information 18(2), 251–289 (2009)
10. Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–102 (1987)
11. Girard, J.Y., Lafont, Y.: Linear logic and lazy computation. In: Ehrig, H., Levi, G., Montanari, U. (eds.) TAPSOFT 1987. LNCS, vol. 250, pp. 52–66. Springer, Heidelberg (1987)
12. Grasbon, D., Braun, N.: A morphological approach to interactive storytelling. In: Proceedings of the Conference on Artistic, Cultural and Scientific Aspects of Experimental Media Spaces (cast01) (2001)
13. Greimas, A.J.: Sémantique structurale: recherche et méthode. Larousse (1966)
14. Gupta, V., Lamping, J.: Efficient linear logic meaning assembly. In: Proceedings of the 17th International Conference on Computational Linguistics (1998)
15. Kakas, A., Miller, R.: A simple declarative language for describing narratives with actions. Journal of Logic Programming 31, 157–200 (1997)
16. Kalvala, S., Paiva, V.D.: Mechanizing linear logic in isabelle. In: 10th International Congress of Logic, Philosophy and Methodology of Science (1995)
17. Lang, R.R.: A declarative model for simple narratives. In: Narrative Intelligence: Papers from the AAAI Fall Symposium. AAAI Press, Menlo Park (1999)
18. Lincoln, P.: Deciding provability of linear logic formulas. In: Advances in Linear Logic, pp. 109–122. Cambridge University Press, Cambridge (1994)
19. Masseron, M.: Generating plans in linear logic: I i. a geometry of conjunctive actions. Theoretical Computer Science 113(2), 371–375 (1993)
20. Masseron, M., Tollu, C., Vauzeilles, J.: Generating plans in linear logic: I. actions as proofs. Theoretical Computer Science 113(2), 349–370 (1993)
21. Miller, R., Shanahan, M.: Narratives in the situation calculus. Journal of Logic and Computation 4, 513–530 (1994)
22. Power, J., Webster, C.: Working with linear logic in coq. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690. Springer, Heidelberg (1999)
23. Propp, V.: Morphology of the Folktale. University of Texas Press (1968)
24. Reiter, R.: Narratives as programs. In: KR. Morgan Kaufmann, San Francisco (2000)
25. Sadrzadeh, M.: Modal linear logic in higher order logic: An experiment with coq. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 75–93. Springer, Heidelberg (2003)
26. Schroeder, M.: How to tell a logical story. In: Narrative Intelligence: Papers from the AAAI Fall Symposium. AAAI Press, Menlo Park (1999)
27. Young, R.M.: Notes on the use of plan structures in the creation of interactive plot. In: Narrative Intelligence: Papers from the AAAI Fall Symposium. AAAI Press, Menlo Park (1999)

Towards Robustness Analysis Using PVS

Renaud Clavel, Laurence Pierre, and Régis Leveugle

TIMA Laboratory (CNRS-INPG-UJF)
46 Av. Félix Viallet - 38031 Grenoble cedex - France
{Renaud.Clavel,Laurence.Pierre,Regis.Leveugle}@imag.fr

Abstract. This work targets the use of formal methods for enhancing dependability analysis of sequential circuits described at the Register Transfer (RT) level. We consider solutions oriented towards theorem-proving techniques as an alternative to classical fault-injection techniques, for analyzing the consequences of errors caused by transient faults. A preliminary study was conducted to evaluate the advantages of a highly automated tool like ACL2 in that context. However, this study showed that, in spite of its numerous advantages, the ACL2 logic is not always expressive enough to deal with the properties under consideration here. In this paper, we briefly explain the shortcomings of ACL2 relatively to our problem, and we investigate the application of PVS, thus enabling to improve our simple and multiple faults models and the associated verification methodology[1].

1 Introduction

The dependability analysis of modern embedded systems is becoming a major concern for designers. Particle strikes, electromagnetic interferences or other signal integrity problems result in soft errors i.e., logic errors in some registers while no damage exists in the circuit. Soft errors can be due to spurious commutations in flip-flops (SEUs - Single Event Upsets - are a well-known example and correspond to single bit-flips) or to erroneous signals latched after propagation in the combinatorial parts (SETs - Single Event Transients). In the following, we focus on *soft errors in random logic parts* i.e., *in flip-flops* (soft errors in memory blocks are often easier to mitigate). Only synchronous digital circuits subject to transient faults, resulting in single or multiple erroneous bits, are considered. No assumption is made on the functionality of the circuit.

The overall goal is to ensure a given level of robustness with respect to faults i.e., either to guarantee that no error (in a specified set of potential errors derived from the selected fault model) can lead to the unwanted events, identified as critical from the application point of view, or to limit the probability of such events to an acceptable value. In order to achieve this goal, designers must analyze at design time the potential consequences of errors, and add additional protections in the circuit when required. Such an analysis is usually based on so-called *fault*

[1] This work has been supported by the French National Research Agency ANR in the framework of the "FME3" project (ANR-07-SESU-006).

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 71–86, 2011.

injection techniques, classically using either simulation or emulation [10]. However such techniques require very long experiment durations, that are often not acceptable in particular in the case of complex circuits and multiple-bit errors. In consequence, current practice is based on partial analyses, injecting only a subset of all possible errors. Furthermore, the number of injected errors is often limited to a very small percentage. Such an approach can be sufficient in some cases to be reasonably confident in the efficiency of some protection mechanisms. However, this cannot be considered as a guarantee that a given dependability property holds for all possible errors in the specified set. Also, it is not possible with such an approach to precisely quantify the probability of a given event; only estimations can be obtained unless exhaustive fault injections are performed. We thus target the development of new methodologies helping the designer in better ensuring that the achieved level of robustness is actually sufficient with respect to the application constraints. Through the use of formal techniques, our objective is a more thorough dependability characterization compared with fault injections using classical approaches.

In the framework of the French FME^3 project [1], a proposed methodology is as follows: deductive proofs (efficient at abstract levels, and also on parameterized systems) can evaluate robustness properties, in particular whether the circuit is fault-tolerant or not, using a meta-model for representing faults. Model-checking based techniques may deliver more information [2]:

- if it is not robust, the computation of the ratios of either potentially or systematically repairing states may give some indications about the robustness of the circuit and may guide the search of the minimal subset of registers to be protected in order to achieve the required robustness,
- if it is robust, the computation of the length of repairing sequences may characterize the repairing speed of the circuit.

Here we describe our early results towards the characterization of fault effects using deductive techniques. We propose to *formally prove that some dependability properties always hold* for a given set of potential errors due to transient faults.

2 Previous Results

2.1 Related Works

Theorem provers provide less automation than algorithmic tools (equivalence checkers, model checkers), but support more powerful logics. Thus they enable the specification of more comprehensive properties, the possibility to reason on complex data types, the ability to consider parameterized devices and properties, while reasoning at various abstraction levels. Surprisingly, to the best of our knowledge, very few actual solutions *based on deduction-oriented approaches* have been proposed in the context of robustness analysis for sequential circuits. Most recent results to apply formal verification to dependability evaluation make use of model checking tools or of symbolic simulation techniques associated with BDD-based or SAT-based solutions.

The approach of [9] focuses on measuring the quality of fault-tolerant designs, and works by comparing fault-injected models with a golden model. The BDD's that correspond to the fault-injected and golden models are built by symbolic simulation for a given number of cycles. Properties that characterize correction capabilities are checked on these models.

The model checker SMV is used in [14] to identify latches that must be protected in an arbitrary circuit. Formal models for all the fault-injected circuits are built (fault injection is performed in one latch for each one of them) and SMV checks whether the formal specification of the original circuit still holds in each case, thus indicating whether the corresponding latch must be protected or not.

The goal of [4] is to analyze the effects of transient faults, using both symbolic simulation and model checking. Injected faults are pictured by modifying the premises of the properties that should be satisfied without faults. Counter examples generated by the model checker are used to interpret the effects of the injected faults.

A definition of the robustness of a circuit in terms of its input/output behaviour is given in [5]. Several fault models are considered, and an algorithm to compute a measure of the robustness is given: it builds a fault-injected model, "unrolls" the circuit and its fault-injected counterpart, and estimates a measure of robustness by SAT-solving equivalence properties.

In [6], the authors propose a HOL formalization of some reliability theory concepts, in which reliability is defined as the probability that the system performs its intended function until some time t (this time to failure is expressed in terms of the failure rate of the system). They focus on the analysis of reconfigurable memory arrays in the presence of fabrication faults such as stuck-at and coupling faults. Our approach concentrates on models for transient faults due for example to particle strikes, and proposes to overcome the limitations of model checkers by means of theorem proving techniques.

2.2 First Solution Using ACL2

In our approach, we make use of a specialized tool called VSYML [11] to get an XML representation of the transition and output functions of a device initially given in VHDL:

$$\delta : I \times S \rightarrow S$$
$$\lambda : I \times S \rightarrow O$$

where I, O and S refer to the sets of input values, output values, and state values (memory elements).

In a first solution [13], we intended to take advantage of the high level of automation of the ACL2 theorem prover [7]. The XML representation was transformed into the appropriate format using a specialized translator. We defined and then formalized in ACL2 the fault model that corresponds to the *presence of a single or multiple-bit error in a single register of the circuit.* This model characterizes the fault-injection function *inject* as a function that satisfies the following conjunction of properties:

- it takes as parameter a state $s \in S$ and returns a state $inject(s) \in S$
- $inject(s)$ is different from s
- only one memorizing element (n-bit register) differs from s to $inject(s)$.

In other words, the injection function belongs to the following set:

$$E = \{inject : S \rightarrow S \mid \forall s \in S, \exists\, !\, i, (inject(s))_i \neq s_i\}$$

Encoding such a formulation in a theorem prover would require the presence of quantifiers and the possibility of specifying functions by characteristic properties instead of using a function definition. Despite the fact that ACL2 is first-order and does not support the explicit use of quantifiers, there are solutions to mimic certain kinds of originally higher-order or quantified statements. For instance, the *encapsulation mechanism* allows to introduce new function symbols that are constrained to satisfy certain axioms, without providing function definitions that uniquely determine the functions's behavior [8]. To guarantee the consistency of the constraints, and thus the soundness of the logical system, a "witness" must be supplied for each constrained function (this witness is a function that can be proven to have the required properties).

Using this principle, it was possible to encode the model above in ACL2 [13]. However the main problem with this implementation is not the use of the "encapsulate" construct, but the fact that the implementation of the third property ("only one state component differs from s to $inject(s)$") in the error model had to be expressed by a theorem of the form: $\bigvee_k (inject(s) = inject_k(s))$ where each $inject_k$ translates an injection in the k^{th} state component.

The drawback of this solution is that the theorem explicitly enumerates every possible error location, thus leading ACL2 to enumerate all the corresponding subgoals. For the small example given as illustration in [13], CPU times remained moderate but they could become prohibitive with more realistic systems.

The conclusion was that it is possible to find a way of encoding our models and associated robustness properties in ACL2, but that it may result in a lack of efficiency for performing proofs.

We thus decided to adopt a less automated proof tool, that provides a more powerful logic. One of the candidates was the PVS proof assistant [12] [3]: its logic and inference mechanisms are powerful enough, without requiring too much effort for realizing proofs; it also supports the definition of strategies that favor proof automation.

3 Fault Models in PVS

We come back to the formalization of fault models. We can take advantage of several PVS features:

- the possibility to parameterize PVS theories,
- the possibility to define predicate functions that make use of $\forall$ and $\exists$ quantifiers,

– the possibility to override a function definition, by means of the WITH construct. The result of an overriden function is exactly the same as the original, except that at the specified arguments it takes the new values.

3.1 Single Faults

The first idea was to characterize the definition of section 2.2 as follows:

$$\forall s, \exists\, i \text{ and } x \text{ such that } x \neq s(i) \text{ and } inject(s) = (s\; WITH\; [i \to x])$$

but we preferred a definition of the form:

$$\forall s, \exists\, i \text{ and } f \text{ such that } inject(s) = (s\; WITH\; [i \to f(s(i))])$$

thus making explicit the fault function f, and enabling the characterization of a parameterized set of injection functions:

$$E(F) = \{inject : S \to S \mid \forall s \in S, \exists\, i, \exists\, f \in F, (inject(s))_i = f(s_i)\}$$

where F is a set of fault functions. As shown thereafter, choosing different instances for this parameter F easily allows considering the same circuit with several fault models for its registers.

This is translated as the following (excerpt of) PVS formalization for faults in a FSM. The last parameter of this theory, *FaultDef*, corresponds to the set denoted F above. In order to distinguish between the symbolic state (in which we assume that there is no injection) and the synthesizable state, we partition the set of states into two components, q and s.

```
FAULT_IN_FSM
  [RegType  : TYPE,   % Possible register types
   RegList  : TYPE,   % Synthesizable state
   SymbList : TYPE,   % Symbolic state
   SelectR  : [RegList  -> [RegType -> bool]],
                      % Types for the registers of the synthesizable state
   SelectS  : [SymbList -> [RegType -> bool]],
                      % Type for the symbolic state
   FaultDef : [i : RegList -> [[(SelectR(i)) -> (SelectR(i))] -> bool]]
                      % Fault model for each state component
  ]
: THEORY BEGIN

  SynthState : TYPE = [i : RegList  -> (SelectR(i))]  % Each state component
  SymbState  : TYPE = [i : SymbList -> (SelectS(i))]  % has a type.
  State : TYPE = [# q : SymbState, s : SynthState #]
                % q is the symbolic state and s is the synthesizable state

  % Predicate for single faults:
  inject_single? (error : [SynthState -> SynthState]) : bool =
    FORALL (s : SynthState):
    EXISTS (i : RegList, f : (FaultDef(i))): error(s) = s with [(i) := f(s(i))]
  % inject_single is the set of functions that satisfy the
  % inject_single? predicate:
  inject_single : TYPE = (inject_single?)
END FAULT_IN_FSM
```

By convention, the transition function δ of each FSM will be referred to as *next* in the PVS code. The theory below specifies the $next^n$ function: for an input sequence ι of length n (i.e., an element of $traces(n)$), a symbolic state q and a synthesizable state s, $next^n$ is defined as follows

$$next^n(\iota_{[0..n-1]}, q, s) = next(\iota_{n-1}, next^{n-1}(\iota_{[0..n-2]}, q, s))$$

and the theorem *REC_thm* can be used to consider $next^n$ in its tail-recursive form when necessary. It states that:

$$next^n(\iota_{[0..n-1]}, q, s) = next^{n-1}(\iota_{[1..n-1]}, next(\iota_0, < q, s >))$$

The corresponding source code is as follows:

```
NEXT_N
  [SymbState : type,
   SynthState : type,
   input : type,
   next : [input, SymbState, SynthState -> [# q : SymbState, s : SynthState #]]
  ]
: THEORY BEGIN

  traces(n : nat) : type = [subrange(0,n-1) -> input]   % input traces of length n

  rec_next(n : nat, i : traces(n), q : SymbState, s : SynthState) :
            RECURSIVE [# q : SymbState, s : SynthState #] =
    if n = 0
    then (# q := q, s := s #)
    else let prev_state = rec_next(n-1, LAMBDA(k:subrange(0,n-2)):i(k), q, s)
         in
             next(i(n-1),prev_state`q,prev_state`s)
    endif
  MEASURE LAMBDA(n : nat, i : traces(n), q : SymbState, s : SynthState): n

  REC_thm: THEOREM
    FORALL (n:nat, i:traces(n), q : SymbState, s : SynthState) :
    n /= 0 IMPLIES
    rec_next(n, i, q, s) =
      let first_state = next(i(0) ,q ,s) in
      rec_next(n - 1, LAMBDA (k:subrange(0,n-2)): i(k+1),
               first_state`q, first_state`s)
END NEXT_N
```

3.2 Spatial Multiplicity

An extension of this model to characterize the *presence of a single or multiple-bit error in several registers of the circuit* is characterized by the following set $E'(k, F)$ of injection functions, where $k \neq 0$ is the number of faulty registers:

$$\begin{aligned} E'(k, F) = \{ & inject : S \rightarrow S \mid \forall s \in S, \exists\, inject_0 \in E(F), \\ & \exists\, inject' \in E'(k-1, F), inject(s) = inject_0(inject'(s))\} \end{aligned}$$

That leads to adding the following piece of code to theory *FAULT_IN_FSM* :

```
% Predicate for possibly multiple faults:
inject_multiple? (k : nat)
               (error : [SynthState -> SynthState]) : RECURSIVE bool =
if (k = 0)
then FORALL (s : SynthState): error(s) = s
else FORALL (s : SynthState):
     EXISTS (f1 : (inject_single?), f2 : (inject_multiple?(k-1))):
                                    error(s) = f1(f2(s))
endif
MEASURE LAMBDA(k : nat): k
% inject_multiple is the set of functions that satisfy the
% inject_multiple? predicate:
inject_multiple(k : nat) : TYPE = (inject_multiple?(k))
```

3.3 Temporal Multiplicity

As far as the temporal multiplicity of faults is concerned, we define a set $E''(k, p, F)$ to characterize the *set of functions that can inject k faults during p cycles*:

– $E''(k, 0, F) = E'(k, F)$
– $E''(0, p, F) = \{\delta^p\}$ where δ is the transition function
– $E''(k+1, p+1, F)$ takes into account both a recursion on E to allow spatial multiplicity, and the use of δ to make time elapse.

4 From VHDL to PVS

4.1 Translation

A toolchain has been created to automatically generate the PVS source code from the original VHDL RTL description (see Fig. 1).

Using our tool VSYML [11], we parse the original VHDL RTL code, perform symbolic execution, and we get an XML representation of the transition and output functions. Templates of robustness theorems (see section 4.2) have been encoded in a library, *xml2pvs*. The intermediate format produced by VSYML is processed, together with the properties to be verified and the patterns of *xml2pvs*, by the tools xsltproc[2] and html2text[3] in order to produce a PVS file that contains both the design description and the theorems to be proven in the presence of faults characterized according to the models of section 3. Proof strategies dedicated to the various types of robustness properties have been defined. They enable the automatization of the proof process.

4.2 Proof Process

Following [1], we consider the notion of "repairing sequences" that are starting from a faulty state and ending in a repairing state, provided no new fault occurs. The predicate $Seq\_rep(s_0, \iota)$ characterizes the fact that (s_0, ι) is a repairing sequence, for a state s_0 and an input sequence ι of length $n+1$:

[2] http://xmlsoft.org/XSLT/xsltproc2.html
[3] http://www.aaronsw.com/2002/html2text/

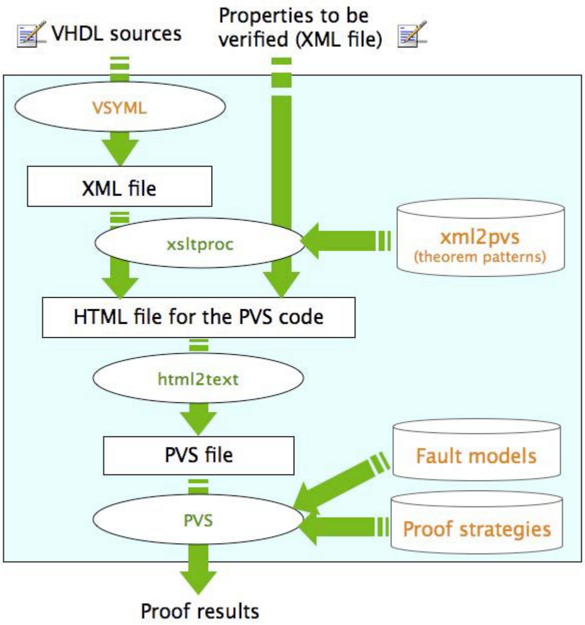


Fig. 1. Toolchain for PVS

$$Seq\_rep(s_0, \iota) = \forall\ \iota' \in traces(j),\ \forall\ e \in E,$$
$$\delta^j(\delta(e(\delta^n(s_0, \iota_{[0..n-1]})), \iota_n), \iota') \in Ref(s_0, \iota, \iota')$$

where $traces(j)$ is the set of traces of length j, E is one of the sets of injection functions of section 3, and $Ref(s_0, \iota, \iota')$ is a reference function. Among the possible reference functions, the most commonly used will be:

$$Ref_{golden}(s_0, \iota, \iota') = \lambda x.\delta^{n+j+1}(s_0, \iota.\iota')R_{equiv}x$$

where R_{equiv} is an equivalence relation. In particular, we can consider the equality R_{equal} and the observational equivalence R_{obs}:

$$R_{equal} = \lambda s.\lambda s'.(s = s')$$
$$R_{obs} = \lambda s.\lambda s'.\forall m \in \mathbb{N}, \forall I \in traces(m), \lambda(\delta^m(s, I)) = \lambda(\delta^m(s', I))$$

In the definition of $Seq\_rep(s_0, \iota)$ above, the term $\delta^j(\delta(e(\delta^n(s_0, \iota_{[0..n-1]})), \iota_n), \iota')$ is the result of injecting an error in the state obtained after n applications of the transition function from s_0, and then applying again the transition function $j + 1$ times. In other words, it corresponds to the state that can be reached $j + 1$ cycles after error injection. If this result belongs to the reference $Ref(s_0, \iota, \iota')$, then (s_0, ι) is said to be a repairing sequence.

The PVS robustness theorems will be of the form:

$$\forall n \in \mathbb{N}, \forall\ \iota \in traces(n + 1), \zeta(\iota)\ \textit{implies}\ (s_0, \iota)\ \textit{is a repairing sequence}$$

where $\zeta(\iota)$ is used, if necessary, to specify hypotheses on the input trace. Together with the notions of repairing sequence and reference function, robustness classes have been defined. The first one (also called strict robustness) states that fault injection does not disturb the circuit behaviour. In that case, $j = 0$ and $Ref(s_0, \iota, \iota')$ is $Ref_{golden}(s_0, \iota, \iota')$. Depending on the equivalence relation R_{equiv}, the corresponding theorems are thus:

$$\begin{aligned}&(1)\ \forall n \in \mathbb{N}, \forall \iota \in traces(n+1), \forall e \in E,\\ &\qquad \zeta(\iota) \Rightarrow \delta(e(\delta^n(s_0, \iota_{[0..n-1]})), \iota_n) = \delta^{n+1}(s_0, \iota)\end{aligned}$$

$$\begin{aligned}&(2)\ \forall n \in \mathbb{N}, \forall \iota \in traces(n+1), \forall e \in E,\\ &\qquad \zeta(\iota) \Rightarrow \forall m \in \mathbb{N}, \forall I \in traces(m),\\ &\qquad\qquad \lambda(\delta^m(\delta(e(\delta^n(s_0, \iota_{[0..n-1]})), \iota_n), I)) = \lambda(\delta^{m+n+1}(s_0, \iota.I))\end{aligned}$$

Another robustness class corresponds to the case where the circuit comes back to a nominal behaviour after a bounded number of cycles k. This gives the following theorems:

$$\begin{aligned}&(1)\ \forall n \in \mathbb{N}, \forall \iota \in traces(n+1), \forall \iota' \in traces(k), \forall e \in E,\\ &\qquad \zeta(\iota) \Rightarrow \delta^k(\delta(e(\delta^n(s_0, \iota_{[0..n-1]})), \iota_n), \iota') = \delta^{n+k+1}(s_0, \iota.\iota')\end{aligned}$$

$$\begin{aligned}&(2)\ \forall n \in \mathbb{N}, \forall \iota \in traces(n+1), \forall \iota' \in traces(k), \forall e \in E,\\ &\qquad \zeta(\iota) \Rightarrow \forall m \in \mathbb{N}, \forall I \in traces(m),\\ &\qquad \lambda(\delta^m(\delta^k(\delta(e(\delta^n(s_0, \iota_{[0..n-1]})), \iota_n), \iota'), I)) = \lambda(\delta^{m+n+k+1}(s_0, \iota.\iota'.I))\end{aligned}$$

PVS proof strategies have been defined for the different types of robustness theorems. On top of them, a strategy *auto-choose* enables the automatization of the proof process, by allowing PVS to automatically choose the adequate proof strategy for the theorem under consideration, based on its identifier. For instance, if the name of the (automatically generated) theorem contains the string “multihardened”, the theorem is about the restoration of a nominal behaviour after multiple faults, and the associated strategy is used. The idea of this strategy is just to split and simplify the goals. If the current goal considers the possibility of having an error in a register R, PVS does not enumerate all the possible states for the other registers. It simply assumes a faulty value in R (according to the fault model), and considers the rest of the registers as a whole.

5 Experiments

We describe here three experiments performed using the approach described above: the cash withdrawal system (ATM) already used as example in [13] [1], a FIR filter, and a CAN interface.

5.1 Cash Withdrawal System

The simple cash withdrawal system (ATM) of Fig. 2 has three synthesized registers that store: the code that is entered through the keyboard (*code*), the valid code (*ok*), and the current number of attempts for entering the code (n).

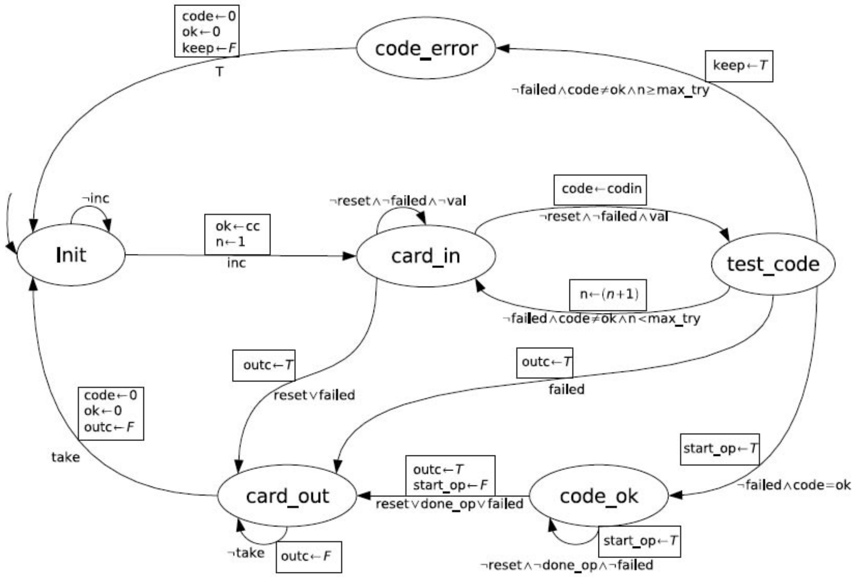


Fig. 2. FSM of the cash withdrawal system

For these 3 registers, through different VHDL configurations, we can choose any register architecture: simple flip-flop, register that detects errors, TMR (triple modular redundancy). In the following, we consider that 3 TMR are used (which means that 3 sub-registers appear for each actual register *code*, *ok* and *n*) and that error injection can occur in any of the synthesized registers.

Here is an excerpt of the PVS source code that corresponds to the ATM and its fault model:

```
ATM_DEF : THEORY BEGIN
tt_localtype_fsmstate : TYPE =
   { init, card_in, test_code, code_error, code_ok, card_out } % FSM states

RegType : DATATYPE BEGIN
   ...
END RegType

RegList : TYPE = { myatm_codereg_mem01, myatm_codereg_mem02,
   myatm_codereg_mem03, myatm_okreg_mem01, myatm_okreg_mem02,
   myatm_okreg_mem03, myatm_nreg_mem01, myatm_nreg_mem02, myatm_nreg_mem03}
SymbList : TYPE = { myatm_atmcontrol_currentstate }

SelectR : [RegList -> [RegType -> bool]] =     % Registers
  LAMBDA(i:RegList) : COND
    i = myatm_codereg_mem01 -> NatReg?,
    i = myatm_codereg_mem02 -> NatReg?,
    i = myatm_codereg_mem03 -> NatReg?,
    % and similarly for the other ones...
  ENDCOND
```

```
SelectS : [SymbList -> [RegType -> bool]] =   % Symbolic state
  LAMBDA(i:SymbList) : COND
    i = myatm_atmcontrol_currentstate -> localtype_fsmstateReg?
  ENDCOND

% Fault model for each register:
FaultDef : [i : RegList -> [[(SelectR(i)) -> (SelectR(i))] -> bool]] =
LAMBDA(i:RegList) : LAMBDA(f : [(SelectR(i)) -> (SelectR(i))]) : COND
  i = myatm_codereg_mem01 -> Natural.inject?( LAMBDA( s : Natural.state ) :
                                                     NatVal(f(NatReg(s)))) ,
  i = myatm_codereg_mem02 -> Natural.inject?( LAMBDA( s : Natural.state ) :
                                                     NatVal(f(NatReg(s)))) ,
  % and similarly for the other ones...
ENDCOND

% Importation of FAULT_IN_FSM with actual parameters:
IMPORTING FAULT_IN_FSM[RegType,RegList,SymbList,SelectR,SelectS,FaultDef] as fsm

input_type : TYPE = [# reset : bool, inc : bool,  cc : nat, codin : nat ,
                      val : bool, doneop : bool, take : bool #]
% Transition function:
next(i : input_type, q : fsm.SymbState, s : fsm.SynthState) : fsm.State = ...
% Output function:
out(i : input_type, q : fsm.SymbState, s : fsm.SynthState) :
     [# outc : bool , keep : bool , start_op : bool , e_detect : bool #] = ...
END ATM_DEF
```

With this configuration, we have the following simple theorem for this ATM example: if the system is in a nominal state and a fault occurs in any of the registers, then the fault is repaired one cycle later. Which is expressed as follows:

```
ATM_Hardened_1 : THEOREM
  FORALL (i : input_type, q : fsm.SymbState, s : fsm.SynthState) :
  FORALL (f : fsm.inject_single):
  ((NatVal(s(myatm_codereg_mem01)) = NatVal(s(myatm_codereg_mem02))))
   and ((NatVal(s(myatm_codereg_mem01)) = NatVal(s(myatm_codereg_mem03))))
   and ((NatVal(s(myatm_okreg_mem01)) = NatVal(s(myatm_okreg_mem02))))
   and ((NatVal(s(myatm_okreg_mem01)) = NatVal(s(myatm_okreg_mem03))))
   and ((NatVal(s(myatm_nreg_mem01)) = NatVal(s(myatm_nreg_mem02))))
   and ((NatVal(s(myatm_nreg_mem01)) = NatVal(s(myatm_nreg_mem03))))
  IMPLIES  next(i,q,f(s)) = next(i,q,s)
```

The proof of this theorem, on an Intel Core2 Duo, takes 43.26 seconds. This is due to the fact that this representation does not take the design hierarchy into account. Therefore we have improved the FSM and fault models to take advantage of hierarchy. For this example, we first prove characteristic properties of the TMR, which takes 1.53 s, then the theorem above is adapted to reflect the presence of this subcomponent, and its proof takes only 12.52 s.

5.2 FIR Filter

A FIR (finite impulse response) filter is a classical function, which behavior is based on a 3-phase computation loop: sample acquisition (e.g., from an ADC converter), computation of the response and result emission on one output. The computation successively adds partial products of n samples with n coefficients. The circuit is made of six sub-blocks: a finite state machine (FSM) for the control of the computation and input/output transfers, a n-position delay line implemented as a FIFO memory (DEC) for storing the last n samples of the signal, a combinatorial multiplier (MULT), a read-only memory storing the coefficients (ROM), an adder with a register for partial product accumulation (ACC) and an output buffer (BUF). More precisely, this FIR filter computes

$$S_t = \sum_{k=0}^{n} I_{t-k} * C_k$$

where the C_k coefficients are stored in the ROM, and the successive inputs I_{t-k} are put into the delay line. We give below an excerpt of the PVS source code for this FIR filter and its fault model. This theory imports the theories related to the sub-components (delay line, ROM, Accu, Buffer, and multiplier).

```
FILTER_DEF  [n : {x : nat | x > 1}, k : nat]
: THEORY BEGIN

tt_localtype_fsmstate : TYPE = { S0, S1, S2, S3 }
IMPORTING DEC[nat, n, 0] as dec1
IMPORTING ROM[nat, n] as rom1
IMPORTING ACCU[nat, 0, LAMBDA(x:nat,y:nat):x+y] as accu1
IMPORTING BUFF[nat, 0] as buff1
IMPORTING MULT as mult1

RegType : DATATYPE BEGIN
  ...
END RegType

RegList : TYPE = { filter1_mydec, filter1_myrom, filter1_myaccu, filter1_mybuff,
                   filter1_mymult}
SymbList : TYPE = { filter1_tapnumber, filter1_cstate}

SelectR : [RegList -> [RegType -> bool]] =       % Registers
  LAMBDA(i:RegList) : COND
    i = filter1_mydec -> DecReg?,
    i = filter1_myrom -> RomReg?,
    i = filter1_myaccu -> AccuReg?,
    i = filter1_mybuff -> BuffReg?,
    i = filter1_mymult -> MultReg?
  ENDCOND

SelectS : [SymbList -> [RegType -> bool]] =     % Symbolic state
  LAMBDA(i:SymbList) : COND
    i = filter1_tapnumber -> IntReg?,
    i = filter1_cstate -> localtype_fsmstateReg?
  ENDCOND
```

```
% Fault model for each register:
FaultDef : [i : RegList -> [[(SelectR(i)) -> (SelectR(i))] -> bool]] =
LAMBDA(i:RegList) : LAMBDA(f : [(SelectR(i)) -> (SelectR(i))]) : COND
  i = filter1_mydec -> dec1.inject_single?
                         (LAMBDA(s:dec1.state): DecValue(f(DecReg(s)))),
  i = filter1_myrom -> FALSE,    % No injection in ROM
  i = filter1_myaccu -> accu1.inject_single?
                         (LAMBDA(s:acu1.state): AccuValue(f(AccuReg(s)))),
  i = filter1_mybuff -> buff1.inject_single?
                         (LAMBDA(s:buff1.state) : BuffValue(f(BuffReg(s)))),
  i = filter1_mymult -> FALSE   % No injection in MULT
ENDCOND
...
END FILTER_DEF
```

An interesting auto-correcting property of the delay line is that *there is no more error in its register if its shift control input has been activated (n+1) times after the occurrence of an error*. To get the proof of this property, we demonstrate various intermediate lemmas about the control part (for the sake of conciseness, these lemmas are not given here), in particular the one that states that $(n + 2)$ cycles are needed between two successive activations of the shift control input of the delay line. More precisely, $(n + 2)$ cycles are needed between leaving the initial state to start the computation of a term and coming back to this initial state. The shift control input can only be activated when the FSM is in the initial state, provided that the primary input *ADC_Busy* is '0'. In the case where we assume that this last condition holds immediately, the theorem above is stated as follows:

```
 FILTER_prop_DEC: THEOREM
   FORALL (l : nat) : l >= ((n+2)*(n+1)) IMPLIES
     FORALL (t : traces(l), q : fsm.SymbState, s : fsm.SynthState) :
     FORALL (f : fsm.inject_single) :
     (FORALL (x : subrange(0,l-1)): t(x)‘adc_busy=FALSE)
     IMPLIES rec_next(l, t, q, f(s)) = rec_next(l, t, q, s)
```

This theorem states that the global state of the FIR filter is sound again after at least $(n + 1) * (n + 2)$ cycles if a fault has been injected in the delay line. It is worth noticing that the proof is parameterized by n, and is performed by induction. It is also independent of the vector sizes. To reduce memory usage, this proof has to be hierarchical: we first prove intermediate lemmas about the DEC (which takes 1.76 s), then the theorem above is proven in 36.13 s.

A generalized version of this theorem can also be obtained. It assumes that at most k cycles are necessary for the environment to put the value '0' on the primary input *ADC_Busy* when the FSM is in its initial state. Then the global state of the FIR filter is sound again after at least $(n+1)*max(n+2, k)$ cycles. In that case, k is another parameter of the inductive proof.

5.3 CAN Interface

The CAN (Controller Area Network) bus protocol is an ISO standard mainly used for automotive applications, but also in industrial automation. The protocol transmits serial frames between several emitters and receivers. The interface used in this study is made of two main modular components: one emitter reading data on a parallel input port, and outputting a serial CAN-frame, and one receiver doing the inverse transform when an incoming frame is identified.

The interest of this example is to demonstrate an appealing usage of the *FaultDef* parameter that easily enables the use of various fault models for the different registers. Here we will concentrate on a simple theorem that states that, *if only the registers of the emitter part can be perturbed (we assume that the other registers are protected), then the behaviour of the receiver will not be disturbed* (for instance it will not accept an erroneous message). The proof of this theorem demonstrates in this case that no shared logic can create common mode failures in the two sub-blocks. We recall that, using *FaultDef*, any injection predicate can be used for each register. Here we use the constant function FALSE for the registers of the receiver part, to indicate that they are protected (thus injection is impossible), and injection functions for integers or bit-vectors for the registers of the transmitter part:

```
FaultDef : [i : RegList -> [[(SelectR(i)) -> (SelectR(i))] -> bool]] =
LAMBDA(i:RegList) : LAMBDA(f : [(SelectR(i)) -> (SelectR(i))]) :
COND
  i = general1_send1_gencrc1_crcreg -> boolVect0_14.inject?(LAMBDA(s :
             boolVect0_14.state) : boolVect0_14Value(f(boolVect0_14Reg(s)))),
  i = general1_send1_sendframe1_bitstuffprec -> Integer.inject?(LAMBDA(s :
             Integer.state) : IntValue(f(IntReg(s)))),
  % and similarly for the other registers of the emitter part...
  i = general1_rec1_rcptframe1_curdata -> FALSE ,
  i = general1_rec1_rcptframe1_curmsg -> FALSE ,
  % and similarly for the other registers of the receiver part...
ENDCOND
```

We are interested in proving that if only the registers of the emitter part can be fault-injected, then the behaviour of the receiver will not be disturbed. It means that the local state of the receiver part is unaffected and will remain unaffected in the next cycles. First, we prove that each register of the receiver part is unaffected, then we prove the following global theorems that state that the global state of the CAN interface may be affected, but by a function g that satisfies the current fault model (i.e., injection is not effective in the receiver), and similarly k cycles later:

```
CAN_MultiPropa : THEOREM
  FORALL (i : input_type, q : fsm.SymbState, s : fsm.SynthState) :
  FORALL (n : nat, f : fsm.inject_multiple(n)):
  EXISTS (m : nat, g : fsm.inject_multiple(m)):
              next(i,q,f(s))`s = g(next(i,q,s)`s)
```

```
CAN_RecMultiPropa : THEOREM
  FORALL (k : nat, i : traces(k)):
  FORALL (q : fsm.SymbState, s : fsm.SynthState):
  FORALL (n : nat, f : fsm.inject_multiple(n)):
  EXISTS (m : nat, g : fsm.inject_multiple(m)):
               rec_next(k,t,q,f(s))`s = g(rec_next(k,t,q,s)`s)
```

Here too, the proofs have to be hierarchical: we first prove intermediate lemmas about the subcomponents (which takes 35.55 s), then the theorems above are proven in 13.26 s and 0.21 s respectively (the proof of the latter uses the former).

6 Conclusion

The main strength of a theorem prover oriented technique is its ability to conduct sophisticated proofs, to reason on various data types and on parametric models. These features are particularly useful in our context of dependability analysis. Moreover, due to the characteristics of the models to be encoded here, choosing a tool like PVS actually brings a solution to our faults and reparation modeling, as illustrated by our experimental results. Future works include improving the automation of the hierarchical decomposition of the translation and proof processes, in order to analyze more complex designs.

References

1. Baarir, S., Braunstein, C., Clavel, R., Encrenaz, E., Ilié, J.M., Leveugle, R., Mounier, I., Pierre, L., Poitrenaud, D.: Complementary Formal Approaches for Dependability Analysis. In: Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2009) (2009)
2. Baarir, S., Braunstein, C., Encrenaz, E., Ilié, J.M., Li, T., Mounier, I., Poitrenaud, D., Younes, S.: Quantifying Robustness by Symbolic Model Checking. In: Proc. Hardware Verification Workshop (July 2010)
3. Crow, J., Owre, S., Rushby, J.M., Shankar, N., Srivas, M.: A Tutorial Introduction to PVS. In: Proc. Workshop on Industrial-Strength Formal Specification Techniques (1995)
4. Darbari, A., Al-Hashimi, B., Harrod, P., Bradley, D.: A New Approach for Transient Fault Injection using Symbolic Simulation. In: Proc. IEEE International On-Line Testing Symposium (2008)
5. Fey, G., Drechsler, R.: A Basis for Formal Robustness Checking. In: Proc. IEEE International Symposium on Quality Electronic Design (2008)
6. Hasan, O., Tahar, S., Abbasi, N.: Formal Reliability Analysis Using Theorem Proving. IEEE Trans. on Computers 59(5) (2010)
7. Kaufmann, M., Manolios, P., Moore, J.: Computer Aided Reasoning: an Approach. Kluwer Academic Pub., Dordrecht (2002)
8. Kaufmann, M., Moore, J.: Structured theory development for a mechanized logic. Journal of Automated Reasoning 26 (2001)

9. Krautz, U., Pflanz, M., Jacobi, C., Tast, H., Weber, K., Vierhaus, H.: Evaluating Coverage of Error Detection Logic for Soft Errors using Formal Methods. In: Proc. DATE 2006 (2006)
10. Leveugle, R., Hadjiat, K.: Multi-level fault injections in VHDL descriptions: alternative approaches and experiments. Journal of Electronic Testing: Theory and Applications 19(5) (October 2003)
11. Ouchet, F., Borrione, D., Morin-Allory, K., Pierre, L.: High-level symbolic simulation for automatic model extraction. In: Proc. IEEE Symposium on Design and Diagnostics of Electronic Systems (2009)
12. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, Springer, Heidelberg (1992)
13. Pierre, L., Clavel, R., Leveugle, R.: ACL2 for the Verification of Fault-Tolerance Properties: First Results. In: Proc. International Workshop on the ACL2 Theorem Prover and Its Applications (2009)
14. Seshia, S., Li, W., Mitra, S.: Verification-guided soft error resilience. In: Proc. DATE 2007 (April 2007)

Verified Synthesis of Knowledge-Based Programs in Finite Synchronous Environments

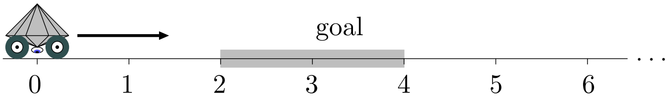
Peter Gammie[1,2]

[1] The Australian National University, Canberra ACT 0200, Australia
Peter.Gammie@anu.edu.au
http://peteg.org/
[2] National ICT Australia

Abstract. Knowledge-based programs (KBPs) are a formalism for directly relating agents' knowledge and behaviour. Here we present a general scheme for compiling KBPs to executable automata with a proof of correctness in Isabelle/HOL. We develop the algorithm top-down, using Isabelle's locale mechanism to structure these proofs, and show that two classic examples can be synthesised using Isabelle's code generator.

1 Introduction

Imagine a robot stranded at zero on a discrete number line, hoping to reach and remain in the goal region $\{2, 3, 4\}$. The environment helpfully pushes the robot to the right, zero or one steps per unit time, and the robot can sense the current position with an error of plus or minus one. If the only action the robot can take is to halt at its current position, what program should it execute?



An intuitive way to specify the robot's behaviour is with this *knowledge-based program* (KBP), using the syntax of Dijkstra's guarded commands:

```
do
    [] $\mathbf{K}_{\text{robot}}$ goal $\rightarrow$ Halt
    [] $\neg\mathbf{K}_{\text{robot}}$ goal $\rightarrow$ Nothing
od
```

where "$\mathbf{K}_{\text{robot}}$ goal" intuitively denotes "the robot knows it is in the goal region" [8, Example 7.2.2]. We will make this precise in §2, but for now note that what the robot knows depends on the rest of the scenario, which in general may involve other agents also running KBPs. In this sense a KBP is a very literal rendition of a venerable artificial intelligence trope, that what an agent does should depend on its knowledge, and what an agent knows depends on what it does. It has

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 87–102, 2011.

been argued elsewhere [4,7,8] that this is a useful level of abstraction at which to reason about distributed systems, and some kinds of multi-agent systems [21]. The downside is that these specifications are not directly executable, and it may take significant effort to find a concrete program that has the required behaviour.

The robot does have a simple implementation however: it should halt iff the sensor reads at least 3. That this is correct can be shown by an epistemic model checker such as MCK [10] or pencil-and-paper refinement [7]. In contrast the goal of this work is to algorithmically discover such implementations, which is a step towards making the work of van der Meyden [18] practical.

The contributions of this work are as follows: §2 develops enough of the theory of KBPs in Isabelle/HOL [19] to support a formal proof of the possibility of their implementation by finite-state automata (§3). The later sections extend this development with a full top-down derivation of an original algorithm that constructs these implementations (§4) and two instances of it (§5 and §6), culminating in the mechanical synthesis of two standard examples from the literature: the aforementioned robot (§5.1) and the muddy children (§6.1).

We make judicious use of parametric polymorphism and Isabelle's locale mechanism [2] to establish and instantiate this theory in a top-down style. Isabelle's code generator [12] allows the algorithm developed here to be directly executed on the two examples. The complete development, available from the Archive of Formal Proofs [9], includes the full formal details of all claims made here.

In the following we adopt the Isabelle convention of prefixing fixed but arbitrary types with an apostrophe, such as $'a$, and suffixing type constructors as in $'a$ list. Other non-standard syntax will be explained as it arises.

2 Semantics of Knowledge-Based Programs

We use what is now a standard account of the multi-agent (multi-modal) propositional logic of knowledge [5,8]. The language of the guards is propositional, augmented by one knowledge modality per agent and parameterised by a type $'p$ of propositions and $'a$ of agents:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{K}_a\ \varphi$$

Formulas are interpreted with respect to a *Kripke structure*, which consists of a set of worlds of type $'w$, an equivalence relation $\sim_a$ for each agent a over these worlds, and a way of evaluating propositions at each world; these are collected in a record of type $('a,\ 'p,\ 'w)$ KripkeStructure. We define satisfaction of a formula φ at a world w in structure M as follows:

$$\begin{array}{lll}
M,\ w \models p & \text{iff} & p \text{ is true at } w \text{ in } M \\
M,\ w \models \neg\varphi & \text{iff} & M,\ w \models \varphi \text{ is false} \\
M,\ w \models \varphi \wedge \psi & \text{iff} & M,\ w \models \varphi \text{ and } M,\ w \models \psi \\
M,\ w \models \mathbf{K}_a\ \varphi & \text{iff} & M,\ w' \models \varphi \text{ for all worlds } w' \text{ where } w \sim_a w' \text{ in } M
\end{array}$$

Intuitively $w \sim_a w'$ if a cannot distinguish between worlds w and w'; the final clause expresses the idea that an agent knows ψ iff ψ is true at all worlds she considers possible (relative to world w)[1]. This semantics supports nested modal operators, so, for example, “the sender does not know that the receiver knows the bit that was sent” can be expressed.

We represent a *knowledge-based program* (KBP) of type ($'a$, $'p$, $'aAct$) KBP as a list of records with fields guard and action, where the guards are knowledge formulas and the actions elements of the $'aAct$ type, and expect there to be one per agent. Lists are used here and elsewhere to ease the generation of code (see §5 and §7). The function set maps a list to the set of its elements.

Note that the robot of §1 cannot directly determine its exact position because of the noise in its sensor, which means that we cannot allow arbitrary formulas as guards. However an agent a *can* evaluate formulas of the form $\mathbf{K}_a\psi$ that depend only on the equivalence class of worlds a considers possible. That φ is a boolean combination of such formulas is denoted by subjective a φ.

We model the agents' interactions using a *finite environment*, following van der Meyden [18], which consist of a finite type $'s$ of states, a set *envInit* of initial states, a function *envVal* that evaluates propositions at each state, and a projection *envObs* that captures how each agent instantaneously observes these states. The system evolves using the transition function *envTrans*, which incorporates the environment's non-deterministic choice of action *envAction* and those of the agents' KBPs into a global state change. We collect these into an Isabelle locale:

```
locale Environment =
  fixes jkbp :: 'a ⇒ ('a, 'p, 'aAct) KBP
    and envInit :: ('s :: finite) list
    and envAction :: 's ⇒ 'eAct list
    and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
    and envVal :: 's ⇒ 'p ⇒ bool
    and envObs :: 'a ⇒ 's ⇒ 'obs
  assumes subj: ∀ a gc. gc ∈ set (jkbp a) ⟶ subjective a (guard gc)
```

A locale defines a scope where the desired types, variables and assumptions are fixed and can be freely appealed to. Later we can instantiate these in various ways (see §4) and also extend the locale (see §3.1).

In the Environment locale we compute the actions enabled at world w in an arbitrary Kripke structure M for each agent using a list comprehension:

```
definition jAction :: ('a, 'p, 'w) KripkeStructure ⇒ 'w ⇒ 'a ⇒ 'aAct list where
  jAction M w a ≡ [ action gc. gc ← jkbp a, (M, w ⊨ guard gc) ]
```

This function composes with *envTrans* provided we can find a suitable Kripke structure and world. With the notional mutual dependency between knowledge and action of §1 in mind, this structure should be based on the set of traces generated by *jkbp* in this particular environment, i.e., the very thing we are in the process of defining. As with all fixpoints there may be zero, one or many

[1] As one would expect there has been extensive debate over the properties of knowledge; the reader is encouraged to consult [8, Chapter 2]. Also their Chapter 7 presents a more general (but non-algorithmic) account of KBPs at a less harried pace.

solutions; the following construction considers a broadly-applicable special case for which unique solutions exist.

We represent the possible evolutions of the system as finite sequences of states, represented by a left-recursive type $'s$ Trace with constructors tInit s and $t \leadsto s$, equipped with tFirst, tLast, tLength and tMap functions.

Our construction begins by deriving a Kripke structure from an arbitrary set of traces T. The equivalence relation on these traces can be defined in a variety of ways [8,18]; here we derive the relation from the *synchronous perfect-recall (SPR) view*, which records all observations made by an agent:

```
definition spr-jview :: $'a \Rightarrow {'s}$ Trace $\Rightarrow {'obs}$ Trace where
  spr-jview $a \equiv$ tMap $(envObs\ a)$
```

The Kripke structure mkM T relates all traces that have the same SPR view, and evaluates propositions at the final state of the trace, i.e., *envVal* ∘ tLast. In general we apply the adjective "synchronous" to relations that "tell the time" by distinguishing all traces of distinct lengths.

Using this structure we construct the sequence of *temporal slices* that arises from interpreting *jkbp* with respect to T by recursion over the time:

```
fun jkbpTn :: nat $\Rightarrow {'s}$ Trace set $\Rightarrow {'s}$ Trace set where
  jkbpT$_0$ $T$ $= \{$ tInit $s\ |s.\ s \in$ set $envInit\ \}$
| jkbpT$_{\mathsf{Suc}\ n}$ $T = \{\ t \leadsto envTrans\ eact\ aact$ (tLast $t$) $|t\ eact\ aact.$
                         $t \in$ jkbpT$_n$ $T \wedge eact \in$ set $(envAction$ (tLast $t$))
                       $\wedge\ (\forall a.\ aact\ a \in$ set (jAction (mkM $T$) $t\ a$)) $\}$
```

We define jkbpT T to be $\bigcup n$ jkbpT$_n$ T. This gives us a closure condition on sets of traces T: we say that T *represents jkbp* if it is equal to jkbpT T. Exploiting the synchrony of the SPR view, we can inductively construct traces of length $n+1$ by interpreting *jkbp* with respect to all those of length n:

```
fun jkbpCn :: nat $\Rightarrow {'s}$ Trace set where
  jkbpC$_0$ $= \{$ tInit $s\ |s.\ s \in$ set $envInit\ \}$
| jkbpC$_{\mathsf{Suc}\ n}$ $= \{\ t \leadsto envTrans\ eact\ aact$ (tLast $t$) $|t\ eact\ aact.$
                         $t \in$ jkbpC$_n \wedge eact \in$ set $(envAction$ (tLast $t$))
                       $\wedge\ (\forall a.\ aact\ a \in$ set (jAction (mkM jkbpC$_n$) $t\ a$)) $\}$
```

We define mkMC$_n$ to be mkM jkbpC$_n$, and jkbpC to be $\bigcup n.$ jkbpC$_n$ with corresponding Kripke structure mkMC.

We show that jAction mkMC $t =$ jAction mkMC$_n$ t for $t \in$ jkbpC$_n$, i.e., that the relevant temporal slice suffices for computing jAction, by appealing to a multi-modal generalisation of the *generated model property* [5, §3.4]. This asserts that the truth of a formula at a world w depends only on the worlds reachable from w in zero or more steps, using any of the agents' accessibility relations at each step. We then establish that jkbpT$_n$ jkbpC $=$ jkbpC$_n$ by induction on n, implying that jkbpC represents *jkbp* in the environment of interest. Uniqueness follows by a similar argument, and so:

Theorem 1. *The set* jkbpC *canonically represents jkbp.*

This is a specialisation of [8, Theorem 7.2.4].

3 Automata for KBPs

We now shift our attention to the problem of synthesising standard finite-state automata that *implement jkbp*. This section summarises the work of van der Meyden [18]. In §4 we will see how these are computed.

The essence of these constructions is to represent an agent's state of knowledge by the state of an automaton (of type $'ps$), also termed a *protocol*. This state evolves in response to the agent's observations of the system using *envObs*, and is deterministic as it must encompass the maximal uncertainty she has about the system. Our implementations take the form of Moore machines, which we represent using a record:

```
record ('obs, 'aAct, 'ps) Protocol =
  pInit :: 'obs ⇒ 'ps     pTrans :: 'obs ⇒ 'ps ⇒ 'ps     pAct :: 'ps ⇒ 'aAct list
```

Transitions are labelled by observations, and states with the set of actions enabled by *jkbp*. The initialising function pInit maps an initial observation to an initial protocol state. A joint protocol *jp* is a mapping from agents to protocols. The term runJP *jp t* runs *jp* on a trace *t* in the standard manner, yielding a function from agents to protocol states. Similarly actJP *jp t* denotes the joint action of *jp* on trace *t*, i.e., $\lambda a.$ pAct (*jp a*) (runJP *jp t a*).

That a joint protocol *jp implements jkbp* is to say that *jp* and *jkbp* yield identical joint actions when run on any canonical trace $t \in$ jkbpC. To garner some intuition about the structure of such implementations, our first automata construction explicitly represents the partition of jkbpC induced by spr-jview, yielding an infinite-state joint protocol:

```
definition mkAuto :: 'a ⇒ ('obs, 'aAct, 's Trace set) Protocol where
  mkAuto a ≡ (| pInit = λobs. { t ∈ jkbpC . spr-jview a t = tInit obs },
                pTrans = λobs ps. { t |t t'. t ∈ jkbpC ∧ t' ∈ ps
                                         ∧ spr-jview a t = spr-jview a t' ↝ obs },
                pAct = λps. jAction mkMC (SOME t. t ∈ ps) a |)
```

```
abbreviation equiv-class a tobs ≡ { t ∈ jkbpC . spr-jview a t = tobs }
```

The function SOME is Hilbert's indefinite description operator ε, used here to choose an arbitrary trace from the protocol state.

Running mkAuto on a trace $t \in$ jkbpC yields the equivalence class of t for agent a, equiv-class a (spr-jview a t), and as pAct clearly prescribes the expected actions for subjective formulas, we have:

Theorem 2. mkAuto *implements jkbp in the given environment.*

3.1 A Sufficient Condition for Finite-State Implementations

van der Meyden showed that the existence of a *simulation* from mkMC to a finite structure is sufficient for there to be a finite-state implementation of *jkbp*

[18, Theorem 2]. We say that a function f, mapping the worlds of Kripke structure M to those of M' is a simulation if it has the following properties:

- Propositions evaluate identically at $u \in \mathsf{worlds}\ M$ and $f\ u \in \mathsf{worlds}\ M'$;
- If two worlds u and v are related in M for agent a, then $f\ u$ and $f\ v$ are also related in M' for agent a; and
- If two worlds $f\ u$ and v' are related in M' for agent a, then there exists a world $v \in \mathsf{worlds}\ M$ such that $f\ v = v'$ and u and v are related in M for a.

From these we have $M,\ u \models \varphi$ iff $M',\ f\ u \models \varphi$ by straightforward structural induction on φ [5, §3.4, Ex. 3.60]. This result lifts through jAction and hence jkbpC_n. The promised finite-state protocol simulates the states of mkAuto.

4 An Effective Construction

The remaining algorithmic obstruction in mkAuto is the appeal to the infinite set of canonical traces jkbpC. While we could incrementally maintain the temporal slices of traces jkbpC_n, ideally the simulated equivalence classes would directly support the necessary operations. We therefore optimistically extend van der Meyden's construction by axiomatising these functions in the SimEnvironment locale of Figure 1, and making the following definition:

definition mkAutoSim :: $'a \Rightarrow ('obs,\ 'aAct,\ 'rep)$ Protocol **where**
  mkAutoSim $a \equiv$
    ⦇ pInit $=$ *simInit a*,
      pTrans $= \lambda obs\ ec.$ (SOME $ec'.\ ec' \in$ set $(simTrans\ a\ ec) \wedge simObs\ a\ ec' = obs)$,
      pAct $= \lambda ec.\ simAction\ ec\ a$ ⦈

The specification of these functions is complicated by the use of *simAbs* to incorporate some data refinement [20], which allows the type *'rep* of representations of simulated equivalence classes (with type *'ss* set) to depend on the entire context. This is necessary because finite-state implementations do not always exist with respect to the SPR view [18, Theorem 5], and so we must treat special cases that may use quite different representations. If we want a once-and-for-all-time proof of correctness for the algorithm, we need to make this allowance here.

A routine induction on $t \in \mathsf{jkbpC}$ shows that mkAutoSim faithfully maintains a representation of the simulated equivalence class of t, which in combination with the locale assumption *simAction* gives us:

Theorem 3. mkAutoSim *implements jkbp in the given environment.*

Note that we are effectively asking *simTrans* to compute the actions of *jkbp* for all agents using only a representation of a simulated equivalence class for the particular agent a. This contrasts with our initial automata construction mkAuto (§3) that appealed to jkbpC for this purpose. We will see in §5 and §6 that our concrete simulations do retain sufficient information.

```
locale SimEnvironment =
      Environment jkbp envInit envAction envTrans envVal envObs
   for jkbp :: 'a ⇒ ('a, 'p, 'aAct) KBP
   and envInit :: ('s :: finite) list
   and envAction :: 's ⇒ 'eAct list
   and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
   and envVal :: 's ⇒ 'p ⇒ bool
   and envObs :: 'a ⇒ 's ⇒ 'obs

   — Simulation operations
+ fixes simf :: 's Trace ⇒ 'ss :: finite
   and simRels :: 'a ⇒ ('ss × 'ss) set
   and simVal :: 'ss ⇒ 'p ⇒ bool

   — Adequacy of representations
   and simAbs :: 'rep ⇒ 'ss set

   — Algorithmic operations
   and simObs :: 'a ⇒ 'rep ⇒ 'obs
   and simInit :: 'a ⇒ 'obs ⇒ 'rep
   and simTrans :: 'a ⇒ 'rep ⇒ 'rep list
   and simAction :: 'rep ⇒ 'a ⇒ 'aAct list

 assumes simf: sim mkMC (mkKripke (simf ‘ jkbpC) simRels simVal) simf
   and simInit:
         ∀ a obs. obs ∈ envObs a ‘ set envInit
               ⟶ simAbs (simInit a obs) = simf ‘ equiv-class a (tInit obs)
   and simObs:
         ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = simf ‘ equiv-class a (spr-jview a t)
               ⟶ simObs a ec = envObs a (tLast t)
   and simAction:
         ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = simf ‘ equiv-class a (spr-jview a t)
               ⟶ set (simAction ec a) = set (jAction mkMC t a)
   and simTrans:
         ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = simf ‘ equiv-class a (spr-jview a t)
               ⟶ simAbs ‘ set (simTrans a ec)
                 = { simf ‘ equiv-class a (spr-jview a (t' ⇝ s)) |t' s.
                      t' ⇝ s ∈ jkbpC ∧ spr-jview a t' = spr-jview a t}
```

Fig. 1. The SimEnvironment locale extends the Environment locale with simulation and algorithmic operations. The backtick ‘ is Isabelle/HOL's image-of-a-set-under-a-function operator. The function mkKripke constructs a Kripke structure from its three components. By *sim M M′ f* we assert that *f* is a simulation from *M* to *M′*.

4.1 A Synthesis Algorithm

We now show how automata that implement *jkbp* can be constructed using the operations specified in SimEnvironment. Taking care with the definitions allows us to extract an executable version via Isabelle/HOL's code generator [12].

We represent the automaton under construction by a pair of maps, one for actions, mapping representations to lists of agent actions, and the other for the transition function, mapping representations and observations to representations. These maps are represented by the types *'ma* and *'mt* respectively, with operations collected in *aOps* and *tOps*. These MapOps records contain *empty*, *lookup* and *update* functions, specified in the standard way with the extra condition that they respect *simAbs* on the domains of interest.

```
abbreviation jkbpSEC ≡ ⋃ a. { simf ‘ equiv-class a (spr-jview a t) |t. t ∈ jkbpC }
```

```
locale Algorithm =
  SimEnvironment jkbp envInit envAction envTrans envVal envObs
                 simf simRels simVal simAbs simObs simInit simTrans simAction
    for jkbp :: 'a ⇒ ('a, 'p, 'aAct) KBP
— ... as for SimEnvironment ...

+ fixes aOps :: ('ma, 'rep, 'aAct list) MapOps
    and tOps :: ('mt, 'rep × 'obs, 'rep) MapOps
  assumes aOps: MapOps simAbs jkbpSEC aOps
      and tOps: MapOps (λk. (simAbs (fst k), snd k)) (jkbpSEC × UNIV) tOps
```

UNIV is the set of all elements of a type. The repetition of type signatures in these extended locales is tiresome but necessary to bring the type variables into scope. As we construct one automaton per agent, we introduce another locale:

```
locale AlgorithmForAgent = Algorithm — ... + fixes a :: 'a
```

The algorithm traverses the representations of simulated equivalence classes of jkbpC reachable via *simTrans*. We use the executable depth-first search (DFS) theory due to Berghofer and Krauss [3], mildly generalised to support data refinement. The DFS locale requires the following definitions, shown in Figure 2:

- an initial automaton *k-empt*;
- the initial frontier *frontier-init* is the partition of the set of initial states under *envObs a*;
- the successor function *k-succs* is exactly *simTrans a*;
- for each reachable state the action and transition maps are updated with *k-ins*; and
- the visited predicate *k-memb* uses the domain of the *aOps* map.

Instantiating the DFS locale is straightforward:

```
sublocale AlgorithmForAgent
      < KBPAlg!: DFS k-succs k-is-node k-invariant k-ins k-memb k-empt simAbs
```

This *conditional interpretation* is a common pattern in these proofs: it says that we can discharge the requirements of the DFS locale while appealing to

```
partial-function (tailrec) gen-dfs where
  gen-dfs succs ins memb S wl = (case wl of
     [] ⇒ S
   | (x·xs) ⇒ if memb x S then gen-dfs succs ins memb S xs
                             else gen-dfs succs ins memb (ins x S) (succs x @ xs))

definition alg-dfs aOps tOps frontier-init simObs simTrans simAction ≡
    let k-empt = (empty aOps, empty tOps);
        k-memb = (λs A. isSome (lookup aOps (fst A) s));
        k-succs = simTrans;
        acts-update = (λec A. update aOps ec (simAction ec) (fst A));
        trans-update = (λec ec′ at. update tOps (ec, simObs ec′) ec′ at);
       k-ins = (λec A. (acts-update ec A, foldr (trans-update ec) (k-succs ec) (snd A)))
     in gen-dfs k-succs k-ins k-memb k-empt frontier-init
```

Fig. 2. The algorithm. The symbol @ denotes list concatenation.

the AlgorithmForAgent context, i.e., the constraints in the SimEnvironment locale and those for our two maps. The resulting definitions and lemmas appear in the AlgorithmForAgent context with prefix KBPAlg.

Our invariant over the reachable state space is that the automaton under construction is well-defined with respect to the *simAction* and *simTrans* functions. The DFS theory shows that the traversal visits all states reachable from the initial frontier, and we show that the set of reachable equivalence classes coincides with the partition of jkbpC under spr-jview a, modulo simulation and representation. Thus the algorithm produces an implementation of *jkbp* for agent a.

We trivially generalise the fixed-agent lemmas to the multi-agent locale:

sublocale Algorithm < KBP!: AlgorithmForAgent — ... a **for** a

The output of the DFS is converted into a protocol using *simInit* and *lookup* on the maps; call this mkAutoAlg. We show in the Algorithm context that mkAutoAlg prescribes the same actions as mkAutoSim for all $t \in$ jkbpC, and therefore:

Theorem 4. mkAutoAlg *is a finite-state implementation of jkbp in the given environment.*

The following sections show that this theory is sound and effective by fulfilling the promises made in the SimEnvironment locale of Figure 1: §5 demonstrates a simulation and representation for the single-agent case, which suffices for finding an implementation of the robot's KBP from §1; §6 treats a multi-agent scenario general enough to handle the classic muddy children puzzle.

5 Perfect Recall for a Single Agent

Our first simulation treats the simple case of a single agent executing an arbitrary KBP in an arbitrary environment, such as the robot of §1. We work in the

$\mathsf{SingleAgentEnvironment}$ locale, which is the $\mathsf{Environment}$ locale augmented with a variable *agent* denoting the element of the $'a$ type. We seek a finite space that simulates mkMC; as we later show, satisfaction at $t \in \mathsf{jkbpC}$ is a function of the set of final states of the traces that *agent* considers possible, i.e., of:

definition $\mathsf{spr\text{-}jview\text{-}abs} :: {'s}\ \mathsf{Trace} \Rightarrow {'s}\ \mathsf{set}$ **where**
  $\mathsf{spr\text{-}jview\text{-}abs}\ t \equiv \mathsf{tLast}\ `\ \mathsf{equiv\text{-}class}\ agent\ (\mathsf{spr\text{-}jview}\ agent\ t)$

To evaluate propositions we include the final state of t in our simulation:

definition $\textit{spr-sim-single} :: {'s}\ \mathsf{Trace} \Rightarrow {'s}\ \mathsf{set} \times {'s}$ **where**
  $\textit{spr-sim-single}\ t \equiv (\mathsf{spr\text{-}jview\text{-}abs}\ t,\ \mathsf{tLast}\ t)$

In the structure mkMCS, $(U,\ u) \sim_a (V,\ v)$ iff $U = V$ and $\textit{envObs agent}\ u = \textit{envObs agent}\ v$, and propositions are evaluated with $\textit{envVal} \circ \mathsf{snd}$. Then:

Theorem 5. mkMCS *simulates* mkMC.

An optimisation is to identify related worlds, recognising that the agent behaves the same at all of these. This quotient is isomorphic to $\mathsf{spr\text{-}jview\text{-}abs}\ `\ \mathsf{jkbpC}$, and so the algorithm effectively simplifies to the familiar subset construction for determinising finite-state automata.

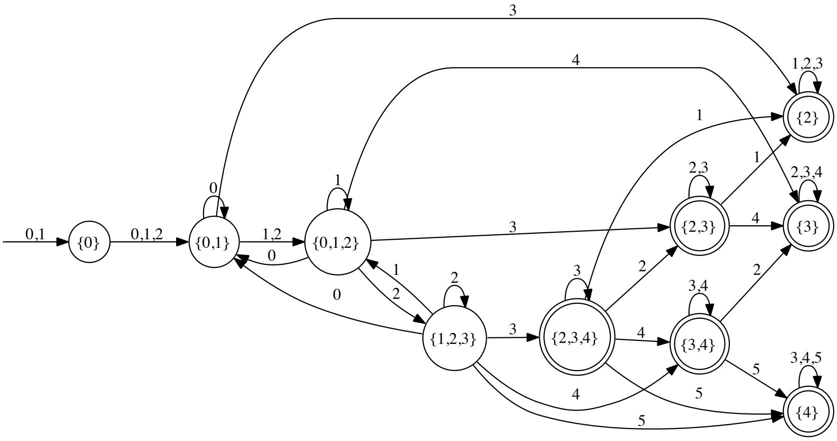
We now address algorithmic issues. As the representations of equivalence classes are used as map keys, it is easiest to represent them canonically. A simple approach is to use *ordered distinct lists* of type $'a\ \mathsf{odlist}$ for the sets and *tries* for the maps. Therefore environment states $'s$ must belong to the class $\mathsf{linorder}$ of linearly-ordered types.

For a set of states X, we define a function $\mathsf{eval}\ X\ \varphi$ that computes the subset of X where φ holds. The only interesting case is that for knowledge: $\mathsf{eval}\ X\ (\mathbf{K}_a\ \psi)$ evaluates to X if $\mathsf{eval}\ X\ \psi = X$, and $\emptyset$ otherwise. This corresponds to standard satisfaction when X represents $\mathsf{spr\text{-}jview\text{-}abs}\ t$ for some $t \in \mathsf{jkbpC}$. The requisite *simObs*, *simInit*, *simAction* and *simTrans* functions are routine, as is instantiating the $\mathsf{Algorithm}$ locale. Thus we have an algorithm for all single-agent scenarios that satisfy the $\mathsf{Environment}$ locale.

A similar simulation can be used to show that there always exist implementations with respect to the multi-agent *clock view* [18, Theorem 4], the weakest synchronous view that considers only the time and most-recent observation.

5.1 The Robot

We now feed the algorithm, the simulated operations of the previous section and a model of the autonomous robot of §1 to the Isabelle/HOL code generator. To obtain a finite environment we truncate the number line at 5. This is intuitively sound for the purposes of determinining the robot's behaviour due to the synchronous view and the observation that if it reaches this rightmost position then it can never satisfy its objective. Running the resulting Haskell code yields this automaton, which we have minimised using Hopcroft's algorithm [11]:



The inessential labels on the states indicate the robot's knowledge about its position, and those on the transitions are the observations yielded by the sensor. Double-circled states are those in which the robot performs the Halt action, the others Nothing. We can see that if the robot learns that it is in the goal region then it halts for all time, and that it never overshoots the goal region. We can also see that traditional minimisation does not yield the smallest automaton we could hope for. This is because the algorithm does not specify what happens on invalid observations, which are modelled as errors instead of don't-cares.

6 Perfect Recall in Broadcast Environments with Deterministic Protocols

We now consider a more involved multi-agent case, where deterministic JKBPs operate in non-deterministic environments and communicate via *broadcast*. It is well known [8, Chapter 6] that simultaneous broadcast has the effect of making information *common knowledge*; roughly put, the agents all learn the same things at the same time as the system evolves, so the relation amongst the agents' states of knowledge never becomes more complex than it is in the initial state.

The broadcast is modelled as a *common observation* of the environment's state that is included in all agents' observations. We also allow the agents to maintain entirely disjoint private states of type *'as*. This is expressed in the locale in Figure 3, where the constraints on *envTrans* and *envObs* enforce the disjointness.

Similarly to §5, we seek a suitable simulation space by considering what determines an agent's knowledge. Intuitively any set of traces that is relevant to the agents' states of knowledge with respect to $t \in \mathsf{jkbpC}$ need include only those with the same common observation as t:

```
record ('a, 'es, 'as) BEState =
  es :: 'es
  ps :: ('a × 'as) odlist — Associates an agent with her private state.

locale DetBroadcastEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: 'a ⇒ ('a :: {finite, linorder}, 'p, 'aAct) KBP
    and envInit :: ('a, 'es :: {finite, linorder}, 'as :: {finite, linorder}) BEState list
    and envAction :: ('a, 'es, 'as) BEState ⇒ 'eAct list
    and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct)
                  ⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState
    and envVal :: ('a, 'es, 'as) BEState ⇒ 'p ⇒ bool
    and envObs :: 'a ⇒ ('a, 'es, 'as) BEState ⇒ ('cobs × 'as option)

+ fixes agents :: 'a odlist
    and envObsC :: 'es ⇒ 'cobs
  defines envObs a s ≡ (envObsC (es s), ODList.lookup (ps s) a)
  assumes agents: ODList.toSet agents = UNIV
    and envTrans: ∀ s s' a eact eact' aact aact'.
          ODList.lookup (ps s) a = ODList.lookup (ps s') a ∧ aact a = aact' a
           ⟶ ODList.lookup (ps (envTrans eact aact s)) a
            = ODList.lookup (ps (envTrans eact' aact' s')) a
    and jkbpDet: ∀ a. ∀ t ∈ jkbpC. length (jAction mkMC t a) ≤ 1
```

Fig. 3. The DetBroadcastEnvironment locale

```
definition tObsC :: ('a, 'es, 'as) BEState Trace ⇒ 'cobs Trace where
  tObsC ≡ tMap (envObsC ∘ es)
```

Unlike the single-agent case of §5, it is not sufficient for a simulation to record only the final states; we need to relate the initial private states of the agents with the final states they consider possible, as the initial states may contain information that is not common knowledge. This motivates the following abstraction:

```
definition  tObsC-abs t ≡ {(tFirst t', tLast t') |t'. t' ∈ jkbpC ∧ tObsC t' = tObsC t}
```

We can predict an agent's final private state on $t' \in$ jkbpC where tObsC t' = tObsC t from the agent's private state in tFirst t' and tObsC-abs t due to the determinacy requirement *jkbpDet* and the constraint *envTrans*. Thus the agent's state of knowledge on t is captured by the following simulation:

```
record ('a, 'es, 'as) SPRstate =
  sprFst :: ('a, 'es, 'as) BEState
  sprLst :: ('a, 'es, 'as) BEState
  sprCRel :: (('a, 'es, 'as) BEState × ('a, 'es, 'as) BEState) set

definition spr-sim :: ('a, 'es, 'as) BEState Trace ⇒ ('a, 'es, 'as) SPRstate where
  spr-sim t ≡ (| sprFst = tFirst t, sprLst = tLast t, sprCRel = tObsC-abs t |)
```

We build a Kripke structure mkMCS of simulated traces by relating worlds U and V for agent a where *envObs a* (sprFst U) = *envObs a* (sprFst V) and *envObs*

a (sprLst U) $=$ *envObs* a (sprLst V), and $\mathsf{sprCRel}$ U $=$ $\mathsf{sprCRel}$ V. Propositions are evaluated by *envVal* $\circ$ sprLst. We have:

Theorem 6. mkMCS *simulates* mkMC.

Establishing this is routine, where the final simulation property follows from our ability to predict agents' private states on canonical traces as mentioned above.

As in §5, we can factor out the common parts of these equivalence classes to yield a denser representation that uses a pair of relations and thus a four-level trie. We omit the tedious details of placating the SimEnvironment locale.

van der Meyden [18, §7] used this simulation to obtain finite-state implementations for non-deterministic KBPs under the extra assumptions that the parts of the agents' actions that influence *envAction* are broadcast and recorded in the system states, and that *envAction* be oblivious to the agents' private states. Therefore those results do not subsume the ones presented here, just as those of this section do not subsume those of §5.

6.1 The Muddy Children

The classic muddy children puzzle [8, §1.1, Example 7.2.5] is an example of a multi-agent broadcast scenario that exemplifies non-obvious reasoning about mutual states of knowledge. Briefly, there are $N > 2$ children playing together, some of whom get mud on their foreheads. Each can see the others' foreheads but not their own. A mother observes the situation and either says that everyone is clean, or says that someone is dirty. She then asks "Do any of you know whether you have mud on your own forehead?" over and over. Assuming the children are perceptive, intelligent, truthful and they answer simultaneously, what will happen?

Each agent child$_i$ reasons with the following KBP:

```
do
    [] $\hat{\mathbf{K}}_{\text{child}_i}\text{muddy}_i$ $\rightarrow$ Say "I know if my forehead is muddy"
    [] $\neg\hat{\mathbf{K}}_{\text{child}_i}\text{muddy}_i$ $\rightarrow$ Say nothing
od
```

where $\hat{\mathbf{K}}_a\varphi$ abbreviates $\mathbf{K}_a\varphi \vee \mathbf{K}_a\neg\varphi$. As the mother has complete knowledge of the situation, we integrate her behaviour into the environment.

In general the determinism of a KBP is a function of the environment, and may be difficult to establish. In this case and many others, however, determinism is syntactically manifest as the guards are logically disjoint, independently of the knowledge subformulas.

The model records a child's initial observation of the mother's pronouncement and the muddiness of the other children in her initial private state, and these states are preserved by *envTrans*. The recurring common observation is all of the children's public responses to the mother's questions. Being able to distinguish these types of observations is crucial to making this a broadcast scenario.

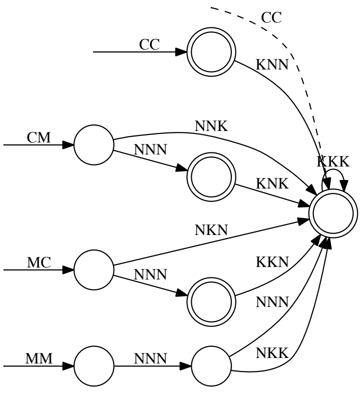


Fig. 4. The protocol of child_0

Running the algorithm for three children and minimising yields the automaton in Figure 4 for child_0. The initial transitions are labelled with the initial observation, i.e., the cleanliness "C" or muddiness "M" of the other two children. The dashed initial transition covers the case where everyone is clean; in the others the mother has announced that someone is dirty. Later transitions simply record the actions performed by each of the agents, where "K" is the first action in the above KBP, and "N" the second. Double-circled states are those in which child_0 knows whether she is muddy, and single-circled where she does not.

To the best of our knowledge this is the first time that an implementation of the muddy children has been automatically synthesised.

7 Perspective and Related Work

The most challenging and time-consuming aspect of mechanising this theory was making definitions suitable for the code generator. For example, we could have used a locale to model the interface to the maps in §4, but as as the code generator presently does not cope with functions arising from locale interpretation, we are forced to say things at least twice if we try to use both features, as we implicitly did in Figure 2. Whether it is more convenient or even necessary to use a record and predicate or a locale presently requires experimentation and perhaps guidance from experienced users.

As reflected by the traffic on the Isabelle mailing list, a common stumbling block when using the code generator is the treatment of sets. The existing libraries are insufficiently general: Florian Haftmann's *Cset* theory[2] does not readily support a choice operator, such as the one we used in §3. Even the heroics of the Isabelle Collections Framework [15] are insufficient as there equality on keys is structural (i.e., HOL equality), forcing us to either use a canonical representation (such as ordered distinct lists) or redo the relevant proofs with reified operations (equality, orderings, etc.). Neither of these is satisfying from the perspective of reuse.

Working with suitably general theories, e.g., using data refinement, is difficult as the simplifier is significantly less helpful for reasoning under abstract quotients, such as those in Figure 1; what could typically be shown by equational rewriting now involves reasoning about existentials. For this reason we have only

[2] The theory *Cset* accompanies the Isabelle/HOL distribution.

allowed some types to be refined; the representations of observations and system states are constant throughout our development, which may preclude some optimisations. The recent work of Kaliszyk and Urban [14] addresses these issues for concrete quotients, but not for the abstract ones that arise in this kind of top-down development.

As for the use of knowledge in formally reasoning about systems, this and similar semantics are under increasing scrutiny due to their relation to security properties. Despite the explosion in number of epistemic model checkers [6,10,13,16], finding implementations of knowledge-based programs remains a substantially manual affair [1]. van der Meyden also proposed a complete semi-algorithm for KBP synthesis [17]. A refinement framework has been developed [4,7].

The theory presented here supports a more efficient implementation using symbolic techniques, ala MCK; recasting the operations of the SimEnvironment locale into boolean decision diagrams is straightforward. It is readily generalised to other synchronous views, as alluded to in §5, and adding a common knowledge modality, useful for talking about consensus [8, Chapter 6], is routine. We hope that such an implementation will lead to more exploration of the KBP formalism.

Acknowledgments. Thanks to Kai Engelhardt for general discussions and for his autonomous robot graphic. Florian Haftmann provided much advice on using Isabelle/HOL's code generator and Andreas Lochbihler illuminated the darker corners of the locale mechanism. The implementation of Hopcroft's algorithm is due to Gerwin Klein. I am grateful to David Greenaway, Gerwin Klein, Toby Murray, Bernie Pope and the anonymous reviewers for their helpful comments on a draft of this paper.

This work was completed while I was employed by the L4.verified project at NICTA. 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 IT Centre of Excellence program.

References

1. Al-Bataineh, O., van der Meyden, R.: Epistemic model checking for knowledge-based program implementation: an application to anonymous broadcast. In: Secure Comm. (2010)
2. 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)
3. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (December 2009) Formal proof development, http://afp.sf.net/entries/Presburger-Automata.shtml
4. Bickford, M., Constable, R.C., Halpern, J.Y., Petride, S.: Knowledge-based synthesis of distributed systems using event structures. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 449–465. Springer, Heidelberg (2005)
5. Chellas, B.: Modal Logic: an introduction. Cambridge University Press, Cambridge (1980)

6. van Eijck, D.J.N., Orzan, S.M.: Modelling the epistemics of communication with functional programming. In: TFP. Tallinn University (2005)
7. Engelhardt, K., van der Meyden, R., Moses, Y.: A program refinement framework supporting reasoning about knowledge and time. In: Tiuryn, J. (ed.) FOSSACS 2000. LNCS, vol. 1784, p. 114. Springer, Heidelberg (2000)
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (1995)
9. Gammie, P.: KBPs. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (May 2011) Formal proof development, http://afp.sf.net/entries/KBPs.shtml
10. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 479–483. Springer, Heidelberg (2004)
11. Gries, D.: Describing an Algorithm by Hopcroft. Acta Informatica 2 (1973)
12. 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)
13. Kacprzak, M., Nabialek, W., Niewiadomski, A., Penczek, W., Pólrola, A., Szreter, M., Wozna, B., Zbrzezny, A.: VerICS 2007 - a model checker for knowledge and real-time. Fundamenta Informaticae 85(1-4) (2008)
14. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: SAC. ACM, New York (2011)
15. Lammich, P., Lochbihler, A.: The isabelle collections framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010)
16. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
17. van der Meyden, R.: Constructing finite state implementations of knowledge-based programs with perfect recall. In: Cavedon, L., Rao, A.S., Wobcke, W. (eds.) PRICAI-WS 1996. LNCS, vol. 1209. Springer, Heidelberg (1997)
18. van der Meyden, R.: Finite state implementations of knowledge-based programs. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180. Springer, Heidelberg (1996)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
20. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press, Cambridge (1998)
21. Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press, New York (2008)

Point-Free, Set-Free Concrete Linear Algebra

Georges Gonthier[*]

Microsoft Research Cambridge
gonthier@microsoft.com

Abstract. Abstract linear algebra lets us reason and compute with collections rather than individual vectors, for example by considering entire subspaces. Its classical presentation involves a menagerie of different set-theoretic objects (spaces, families, mappings), whose use often involves tedious and non-constructive pointwise reasoning; this is in stark contrast with the regularity and effectiveness of the matrix computations hiding beneath abstract linear algebra. In this paper we show how a simple variant of Gaussian elimination can be used to model abstract linear algebra directly, using matrices only to represent all categories of objects, with operations such as subspace intersection and sum. We can even provide effective support for direct sums and subalgebras. We have formalized this work in Coq, and used it to develop all of the group representation theory required for the proof of the Odd Order Theorem, including results such as the Jacobson Density Theorem, Clifford's Theorem, the Jordan-Holder Theorem for modules, the Wedderburn Structure Theorem for semisimple rings (the basis for character theory).

Keywords: Formalization of Mathematics, Linear Algebra, Module Theory, Algebra, Type inference, Coq, SSReflect.

1 Introduction

General linear algebra[1] is amongst the most ubiquitous and useful basic nontrivial mathematical theory, probably because it mediates calculations and combinatorial deductive reasoning, linking computations in cartesian coordinates to abstract geometric arguments, or purely combinatorial properties of finite groups with algebraic properties of their linear representations. Developing a good linear algebra library was one of the important side goals of our Feit-Thompson Theorem proof project[2,3,4,5,6,7].

Naturally, most computer proof systems supply one (or more!) linear algebra libraries[8,9,10,11,12,13]. However most are limited to the algebra of vectors and/or matrices and do not support point-free reasoning using whole subspaces. The rare exceptions[10,12,14] use classical sets to represent subspaces. This basically combinatorial account fails to capture some specifics of linear subsets, in particular their algebraic properties under sum, intersection and linear image.

Note however that all objects used in linear algebra can be represented as matrices: endomorphisms by their matrix, (row) vectors by $1 \times n$ matrices, lists

[*] This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 103–118, 2011.

of vectors and bases by rectangular matrices, and subspaces by a basis. Under this identification the same matrix multiplication operation AF can mean composing A and F, applying F to A, mapping F over A or taking the image of A under F. The (unique) matrix product associativity and distributivity laws are consistent with all those interpretations — a major simplification of the theory.

We came to this observation by accident. Because we wanted a constructive formalization of linear algebra, we had to define an effective membership test for linear sets. After working out a suitable generalization of Gaussian elimination we realized it actually provided all the set theoretic subspace constructions, so we could do away with the entire set-theoretic boilerplate and use matrices only.

We then applied the resulting library to one of the then outstanding prerequisites of the Feit-Thompson Theorem — an extensive development of group module and representation theory. This worked out remarkably well, and was also invaluable in shaping the details and ironing out all the kinks of the core linear algebra formalization, for instance prompting the development of indexed subspace sums and directed sums.

It is our experience that such large scale use is essential for obtaining a usable formalization. With an appropriate framework, all basic linear algebra proofs are trivial (2-5 lines) and hence provide no useful feedback on the library design choices. Linear subspace theory is in the 10-line range and similarly offers little guidance. It is only with representation theory, with proofs in the 30-50 line range, that we started to identify substantial issues, and the hardest issues, such as the need to support complex direct sums and non-constructive results, only appeared in the Feit-Thompson Theorem proof itself, with proofs in the 200+ line range.

The contributions of this paper are thus: a practical matrix encoding of linear subspaces and their operations (section 3), an innovative use of type inference and dependent types to formalize general direct sums of subspaces (section 4), and a large-scale validation of the resulting library with an extensive library on finite group representations and its application to the Local Analysis part of the Feit-Thompson Theorem proof[3] (section 5).

This work was done using the SSReflect extension of the Coq system[15,16]. We review the basic SSReflect matrix algebra library [16,6] in section 2, and use mathematical notation as much as possible in section 3, but due to lack of space we assume some familiarity with our prior work[7,6] in the more technical sections 3.4 and 4.

The libraries described here can be viewed at `http://coqfinitgroup.gforge.inria.fr/`; they will be distributed as part of the next SSReflect release, early during the review period.

2 Matrix Operations

2.1 A Combinatorial and Algebraic Hierarchy

Matrices are a typical *container* type. The properties of a given matrix type will typically be a function of the properties of the type of its coefficients: while all matrices will share some structural properties such as shape, only matrices with comparable elements can be compared, only matrices over a ring can be multiplied, etc.

In the SSReflect library this notion of "type with properties" is captured with Structures, which are just (higher-kinded) record types with two fields, a *sort*, or carrier type, and a *class*, itself a record providing various operations over the sort along with some of their properties. For example, a "comparable" type, or eqType, could be described as follows:

```
Module Equality.
Record class_of (T : Type) : Type :=
  Mixin {op : T -> T -> bool; _ : forall x y, x = y <-> op x y}.
Structure type : Type := Pack {sort; class : class_of sort}.
End Equality.
Notation eqType := Equality.type.
Coercion Equality.sort : eqType >-> Sortclass.
Definition eq_op T := Equality.op (Equality.class T).
Notation "x == y" := (@eq_op _ x y).
```

The Coercion line lets us use an eqType as a type, as in

```
Let swap {T : eqType} (x y z : T) := if z == x then y else z.
```

Note that this is very similar to the Haskell type class mechanism, except for the extra layer of packaging introduced by the type Structure, which is made possible by Coq's higher-kinded types. This extra packaging has important consequences on the feasibility of type checking, especially in the presence of container types such as matrices[7].

Similarly to the type class Instance declaration, the Canonical Structure declaration lets us tie specific structures to existing types, e.g., allowing us to equip bool and nat with definitions for _ == _.

The SSReflect library defines many such structures (97 at last count), which provide many standard sets of operations, from basic combinatorial fare such as eqType above to standard algebraic objects such as rings and fields, and combinations thereof such as finite fields. Here are a few of the less common ones

- finType a finite, explicitly enumerable type; any subset A of a finType can be enumerated (enum A) and counted (#|A|).
- choiceType a type with a choice operator choose P x_0 that picks a *canonical* x such that Px holds, given x_0 such that Px_0.
- zmodType a type with an addition operation, and therefore integer scaling.
- lmodType R a type with both an addition operation and a scaling operation (denoted α*:v in Coq) with coefficients α in R (which must be a ringType).
- unitRingType a ring with an effective test for unit (invertible) elements, and a partial inverse function for its units.

These Structures are arranged in a (multiple) inheritance hierarchy in the obvious way[7]. It is important to note that zmodType, the smallest algebraic structure, inherits from both eqType and choiceType.

Let us finally point out that unlike Haskell type classes (but similarly to their Coq reinterpretation[17], Structure keys are not limited to types. The "big operator" library[6] uses these to recognize AC operators, and we will be using below similar structures to quantify over linear functions (between lmodTypes) and ring morphisms.

2.2 Basic Algebra

Matrices are basically tabulations of functions with a finite rectangular domain of the form $[0, m) \times [0, n)$. The SSReflect library defines both finite index types (`ordinal n`, denoted `'I_n`), and a generic tabulation type constructor `{ffun ..}` for functions with a `finType` domain, which we simply combine:

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Note that `matrix R m n` is a *dependent* type that is specialized to the $m \times n$ shape. This is required to develop an algebraic theory, because too many laws do not generalize to "matrices of unknown shape". The usual Coq notation for this type is `'M_(m, n)` as `R` can usually be inferred from context.

The `finfun` library provides us with a one-to-one correspondence between `{ffun A -> R}` and `A -> R`, which we only need to curry to get a

```
Coercion fun_of_matrix : matrix >-> Funclass.
```

that lets us write `A i j` in Coq for the A_{ij} coefficient of A. We provide a dual notation for defining matrices, which we use for all matrix arithmetic operators:

```
Definition addmx A B := \matrix_(i, j) (A i j + B i j).
Definition mulmx A B := \matrix_(i, k) \sum_j (A i j * B j k).
```

(We have omitted some type declarations.) While they may not use the most efficient algorithms, these definitions have the advantage of actually being *useful* for proving algebraic identities. Indeed most algebraic identities can be proved in one or two lines.

We then declare `Canonical z/lmodType Structure`s so that addition and scaling can be denoted with the generic `+` and `*:` operators, and all lemmas of the generic algebra package become available. However, we still require a separate operator (denoted `*m`) for multiplication, because only nontrivial square matrix types are proper `ringType`s.

This is about the point where most matrix libraries end, but we can easily carry on and define the `unitRingType` structure, along with determinants, cofactors, and adjugate matrices, by leveraging the SSReflect permutation library[5]:

```
Definition determinant n (A : 'M_n) : R :=
  \sum_(s : 'S_n) (-1) ^+ s * \prod_i A i (s i).
Definition cofactor n A (i j : 'I_n) : R :=
  (-1) ^+ (i + j) * determinant (row' i (col' j A)).
Definition adjugate n (A : 'M_n) := \matrix_(i, j) cofactor A j i.
```

Even these proofs remain relatively easy: it takes about 20 lines to show the Cauchy determinant product formula $|AB| = |A|.|B|$ and the Laplace expansion formula for cofactors, and then 9 lines to derive the Cramer rule $A.(\mathbf{adj}\ A) = |A|.1$, from which we can prove the Cayley-Hamilton theorem in three lines[6].

2.3 Block and Reshaping Operations

Matrices are also combinatorial objects, and the SSReflect `matrix` library supplies some 26 operations for rearranging the contents of matrices over *any* type. This includes transposition and extraction, permutation and suppression of row and columns (the `row'` and `col'` functions above perform the latter). Most importantly, this also includes operations for cutting and pasting *block matrices*, e.g.,

```
Definition row_mx (A : 'M_(m, n)) (B : 'M_(m, p)) : 'M_(m, n + p)
  := \matrix_(i, j)
      match split j with inl k => A i k | inr l => B i l end.
```

computes the block row matrix $(A\ B)$, using the function

```
split : 'I_(n + p) -> 'I_n + 'I_p
```

from the `fintype` library to map column indices to the appropriate submatrix. A set of lemmas extends the usual matrix computation rules to 1×2, 2×1 and 2×2 block matrices, which let us prove many identities without having to consider individual coefficients.

As with the call to `split` above, it is usually not necessary how a block matrix is subdivided — the syntactic shape of the dimensions supplies that information via type inference. There is a downside: we may end up with matrices that have extensionally, but not syntactically the correct shape, for instance when stating block matrix associativity. We use a *cast* operation to mitigate this:

```
   castmx : (m = m') * (n = n') -> 'M_(m, n) -> 'M_(m', n').
Lemma row_mxA : forall m n1 n2 n3 A B C,
  let cast := (erefl m, esym (addnA n1 n2 n3)) in
  row_mx A (row_mx B C) = castmx cast (row_mx (row_mx A B) C).
```

Note that `castmx` is bidimensional; its first argument is proof-irrelevant (because `nat` is an `eqType`) so we can prove rewrite rules that make it easy to move, collect and eliminate casts. We also provide a prototype-based cast: `conform_mx` A B returns a matrix that has syntactically the same shape as A, but is equal to B if B has extensionally the same shape as A (and A otherwise).

Finally we define reshaping operations `mxvec` and `vec_mx` that turn a rectangular $m \times n$ matrix into linear $1 \times mn$ row vector and conversely.

3 Gaussian Elimination and Row Spaces

Here we show how to develop an algorithmic theory of linear algebra on the basis of a single Gaussian elimination procedure. We shall assume that all our matrices are over a fixed field.

3.1 Extended Gaussian Elimination

All that is needed to extend Gaussian elimination gracefully to singular matrices is to perform *double pivoting*, i.e., to search for a non-zero pivot in all the matrix and then swap both rows and columns to bring it to the top left corner. This gives an exact value for the rank of the matrix, as the decomposition stops exactly when it reaches a null matrix. This is our Coq code for this algorithm, which can also be viewed as a degenerate, easy case of the Smith normal form computation[1].

```
Fixpoint gaussian_elimination {m n} :=
  match m, n return 'M_(m, n) -> 'M_m * 'M_n * nat with
  | _.+1, _.+1 => fun A : 'M_(1 + _, 1 + _) =>
    if [pick ij | A ij.1 ij.2 != 0] is Some (i, j) then
      let a := A i j in let A1 := xrow i 0 (xcol j 0 A) in
      let u := ursubmx A1 in let v := a^-1 *: dlsubmx A1 in
      let: (L, U, r) := gaussian_elimination (drsubmx A1 - v *m u)
      in (xrow i 0 (block_mx 1 0 v L),
          xcol j 0 (block_mx a%:M u 0 U),
          r.+1)
    else (1%:M, 1%:M, 0%N)
  | _, _ => fun _ => (1%:M, 1%:M, 0%N)
  end.
```

This is virtually identical to the LUP decomposition procedure described in [7], and its correctness is easily established in a similar manner. Besides the double pivoting, the only differences are that row and column permutations are combined with the lower and upper triangular factors of the decomposition, and that the decomposition of a null matrix is a pair of identity matrices (`1%:M` is our Coq notation for a scalar matrix with 1s on the diagonal). If we denote by 1_r a (not necessarily square) matrix that has 1s in r first coefficients on the main diagonal and 0 elsewhere

$$1_r = \begin{pmatrix} {}^{1}\ddots{}_{1} & 0 \\ 0 & 0 \end{pmatrix}$$

and set `gaussian_elimination` $A = (A_{\hat{C}}, A_{\hat{R}}, r(A))$, then the correctness of the above function is expressed by the five conditions

$$r(A) \leq m, n \qquad A_{\hat{C}}, A_{\hat{R}} \text{ invertible} \qquad A_{\hat{C}}\, 1_{r(A)}\, A_{\hat{R}} = A$$

We call $A_{\hat{C}}$ and $A_{\hat{R}}$ the extended column and row bases of A, respectively. The column (resp. row) basis A_C (resp. A_R) of A consists or the $r(A)$ first columns (resp. rows) of $A_{\hat{C}}$ (resp. $A_{\hat{R}}$). Since $1_{r(A)} 1_{r(A)} = 1_{r(A)}$ we have

$$A_C = A_{\hat{C}}\, 1_{r(A)} \quad A_R = 1_{r(A)}\, A_{\hat{R}} \quad A_C\, A_R = A$$

3.2 Rank Theory

The fact that $r(A)$ has indeed the properties of the matrix rank follows directly from the correctness conditions above and from the following two basic facts about matrices over a commutative ring:

Lemma 1. *If A and B are respectively $m \times n$ and $n \times m$ matrices such that $AB = 1$, then $m \leq n$, and if $m = n$ then $BA = 1$.*

We prove the second assertion first. Consider $A' = |B|.(\mathbf{adj}\ A)$; then

$$A'A = |B|.(\mathbf{adj}\ A)A = |B|.|A|.1 = (|A||B|).1 = |AB|.1 = 1$$

so $BA = A'ABA = A'A = 1$. For the first assertion, assume $n < m$, and let $A = (A_l\ A_r)$ where A_l is a square $n \times n$ matrix, and similarly $B = \left(\begin{smallmatrix} B_u \\ B_d \end{smallmatrix}\right)$ with B_u square. Block product now gives

$$AB = (A_l\ A_r)\begin{pmatrix} B_u \\ B_d \end{pmatrix} = \begin{pmatrix} A_lB_u & A_rB_u \\ A_lB_d & A_rB_d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

so $A_lB_u = 1$, whence $B_uA_l = 1$ by the second part, and so $1 = A_rB_d = A_rB_uA_lB_d = 0$, a contradiction since this is a nontrivial $(m-n) \times (m-n)$ matrix.

This yields both upper and lower bounds on the rank function:

Lemma 2. *If M and N are respectively $m \times r$ and $r \times n$ matrices, then $r(MN) \le r$, and any A such that $NAM = 1$ must have $r(A) \ge r$.*

For the first assertion, let $L = 1_{r(MN)}(MN)^{-1}_{\hat{C}}$ and $U = (MN)^{-1}_{\hat{R}}1_{r(MN)}$ be respectively $r(MN) \times m$ and $n \times r(MN)$ matrices, and apply the first lemma to

$$(LM)(NU) = L(MN)_{\hat{C}}1_{r(MN)}(MN)_{\hat{R}}U = 1_{r(MN)}11_{r(MN)}11_{r(MN)} = 1$$

For the second assertion, apply the lemma to $(NA_C)(A_RM) = NAM = 1$.

It then follows immediately that $r(AB) = r(A_CA_RB_CB_R) \le r(A), r(B)$ and

$$r(A+B) = r(A_CA_R + B_CB_R) = r\left((A_C\ B_C)\begin{pmatrix} A_R \\ B_R \end{pmatrix}\right) \le r(A) + r(B)$$

3.3 Set Operations

The extended row and column bases provide everything we need to implement set-theoretic operations on matrices. We define

$$\begin{array}{rl} \mathbf{ker}\, A = & (1 - 1_{r(A)})A^{-1}_{\hat{C}} \\ \mathbf{coker}\, A = & A^{-1}_{\hat{R}}(1 - 1_{r(A)}) \\ A \le B \Leftrightarrow & A(\mathbf{coker}\, B) = 0 \\ A \equiv B \Leftrightarrow & A \le B \le A \end{array} \qquad \begin{array}{rl} A^{\sim 1} = & A^{-1}_{\hat{R}}1_{r(A)}A^{-1}_{\hat{C}} \\ A +_s B = & \left(\begin{smallmatrix} A \\ B \end{smallmatrix}\right) \\ A \cap_s B = & [\mathbf{ker}\,(A +_s B)]_l A \\ \overline{A} = & (1 - 1_{r(A)})A_{\hat{R}} \end{array}$$

We can see that $\mathbf{ker}\, A$ is the kernel of A viewed as a linear function since $(\mathbf{ker}\, A)A = (1 - 1_{r(A)})1_{r(A)}A_{\hat{R}} = 0$, and likewise that $\mathbf{coker}\, A$ is the cokernel of A. It follows that $A \le B$ tests whether the row space of A is included in that of B, i.e., whether A (considered as a subspace) is included in B, and that $A \equiv B$ tests whether A and B represent the same subspace. $B^{\sim 1}$ is a partial inverse to B, since, if $A \le B$ we have

$$A - AB^{\sim 1}B = AB^{-1}_{\hat{R}}B_{\hat{R}} - AB^{-1}_{\hat{R}}1_{r(B)}B_{\hat{R}} = A(\mathbf{coker}\, B)B_{\hat{R}} = 0$$

Thus $A \le B$ if and only if $A = DB$ for some matrix D. Note finally that if v is a row vector, then $v \le B$ tests whether v is in the row space of B.

Obviously the row space of $A +_s B$ is the sum of the row spaces of A and B. In the definition of $A \cap_s B$, $K = \mathbf{ker}\,(A +_s B)$ is a square block matrix which we divide vertically; K_l designates the left (rectangular) block. To see that $A \cap_s B$ is indeed the intersection of A and B observe that

$$0 = K \begin{pmatrix} A \\ B \end{pmatrix} = K_l A + K_r B$$

so indeed $K_l A = -K_r B \leq A, B$. Conversely if $C = A'A = B'B$, then $(A' - B')(A +_s B) = 0$ so $(A' - B') \leq K$ and $C = A'A = DK_l A$ for some D, hence $C \leq A \cap_s B$.

All of the usual results on linear spaces and bases easily follow from these definitions, as well as some more advanced ones like the Frobenius rank inequality

$$r(AB) + r(BC) \leq r(B) + r(ABC)$$

all with proofs under twelve lines (most are under two) and *no induction*. The reason for this is that all the induction we need is neatly encapsulated inside the Gaussian elimination procedure. Indeed it is instructive to consider why $\overline{A}$, defined above as $A_{\hat{R}}$ with the top $r(A)$ rows zeroed out, is indeed a complement to the row space of A. The nonzero rows of $\overline{A}$ are the rows of the identity matrix returned by the base case of `gaussian_elimination`, permuted by the pivot column transpositions during the unwinding of the recursion. Thus these are vectors of the standard basis that complete the row base of A: our seemingly trivial changes to the LUP decomposition algorithms are in fact a proof of the incomplete basis theorem.

3.4 Algebras and Subrings

The next structure up from linear spaces are F-algebras, which add a multiplicative ring structure. A finite dimensional F-algebra can always be embedded in its algebra of endomorphisms, which is a matrix algebra, so we ought to be able to extend our program of “doing it all with matrices” to F-algebras as well. However, there is a catch. To enjoy the natural ring structure of matrices, algebra elements should be square matrices; but to be considered as points in our encoding of subspaces, they should be flat row vectors.

Our solution is to stick to square matrices, but to use the reshaping function `vecmx` of section 2.3 when we need to test for membership in a subalgebra:

$$(f \in R) \Leftrightarrow (\texttt{mxvec}\ f \in R)$$

Note that if f is an $n \times n$ matrix, then R will have to be an $m \times n^2$ matrix (whose type will be denoted `'A_(m,n)`). The pointwise product of two subalgebras can be defined using the iterated sums of normalized spaces we will define in section 4.2

```
Definition mulsmx m1 m2 n (R1 : 'A_(m1, n)) (R2 : 'A_(m2, n)) :=
  (\sum_i <<R1 *m lin_mx (mulmxr (vec_mx (row i R2)))>>)%MS.
```

We used the `lin_mx` function to tabulate a linear matrix-to-matrix function:

```
Definition lin_mx (f : 'M[R]_(m1, n1) -> 'M[R]_(m2, n2)) :=
  \matrix_(i, j) mxvec (f (vec_mx (delta_mx 0 i))) 0 j.
```

Further, by combining lin_mx with our set-like linear functions we can define ideals, subrings, centralisers and centers of algebras, as we can program effective tests for just about any linear condition. For example we can test whether a subalgebra R has an identity element (an $e \neq 0$ such that $ef = fe = f$ for all $f \in R$) with the following predicate

```
Definition has_mxring_id m n (R : 'A_(m , n)) :=
  (R != 0) &&
  (row_mx 0 (row_mx (mxvec R) (mxvec R))
    <= row_mx (cokermx R)
      (row_mx (lin_mx (mulmx R \o lin_mulmx))
             (lin_mx (mulmx R \o lin_mulmxr))))%MS.
```

The lower inclusion is satisfied iff there is an e such that the left-hand side is equal to the product of $v =$ mxvec e by the right-hand side, i.e., that $u(\textbf{coker}\ R) = 0$ and $R = R($lin_mx (mulmx e)$) = R($lin_mx (mulmxr e)$)$, that is, $e \in R$ and $ef = fe = f$ for all the $f =$ vec_mx (row i R), which generate R.

4 General Direct Sums

The concept of *direct sum* is one of the more powerful tools for reasoning about collections of subspaces, because it links a strong combinatorial property (unique decomposition) to a simple arithmetic (in)equality of ranks. This correspondence is especially useful when applied to general iterated sums, but there are some intricate technical issues that must be addressed to formalize it in Coq.

4.1 On Subspace Equality

While the theory exposed in Section 3 lets us compute and reason with subspaces represented as matrix row spaces, it does not provide a unique representation for subspaces. Indeed, for any given A, there are many $B \equiv A$, and this remains true even if we restrict ourselves to square matrices.

In addition, the "setoid" framework that implements relational congruence in Coq[18,15] is incapable of dealing with the multiply dependent, polymorphic relation $A \equiv B$. We must resort to a proxy relation A :=: B that lets us replace A by B directly in expressions of the form $r(A)$, $A \leq C$ and $C \leq A$; these three cases cover most of the contexts in which we need to substitute equivalent subspace expressions.

For other contexts, we can either compose context lemmas directly or use the choiceType structure to obtain a *standard* representation:

$$\langle A \rangle = \texttt{choose}\ (\lambda B : M_n.\ A \equiv B)\ (1_{r(A)} A_{\hat{R}})$$

This defines $\langle A \rangle$ (Coq notation: <<A>>) as a square matrix with the same row space as A, such that $\langle A \rangle = \langle B \rangle$ iff $A \equiv B$.

4.2 Monoidal Set Operations

While the SSReflect bigop library[6] will let us turn the binary subspace operators $+_s$ and $\cap_s$ into n-ary ones, most of its facilities would be unusable because they require strictly monoidal operators (e.g., we need $A +_s 0 = A$, not $A +_s 0 \equiv A$). Fortunately, it turns out we can use $\langle A \rangle$ to fix this, by setting:

$$A +_{ss} B = \begin{cases} A & \text{if } B = 0 \text{ and } A \text{ is square} \\ B & \text{if } A = 0 \text{ and } B \text{ is square} \\ \langle A +_s B \rangle & \text{otherwise} \end{cases}$$

We use the `conform_mx` function of section 2.3 to code the first two cases in Coq. It is easy to show that $+_{ss}$ is strictly monoidal as its identity element 0 is only equivalent to one square matrix — itself. Thus, this definition lets us use generic `bigop` sums for subspace sums (Coq notation `(\sum_ ...)%MS`).

Obtaining a strictly monoidal intersection is similar but more delicate because although we can choose the identity matrix 1 as the identity element, it is by no means unique. We need to ensure that our normalization operation does not return 1 by accident; we thus write $A \simeq 1$ when $A \equiv 1$ and $A = 1$ if A is square, and let $\langle A \rangle_1$ be a canonical square matrix $B \equiv A$ such that $B = 1$ iff $A \simeq 1$. Then we can take $A \cap_{ss} B$ to be B if $A \simeq 1$ and B is square, A if $B \simeq 1$ and A is square, else $\langle A \rangle_1$ if $B \equiv 1$, and $\langle A \cap_s B \rangle_1$ otherwise.

4.3 A Direct Sum Package

A binary sum $A +_s B$ is direct iff $A \cap_s B = 0$, or, equivalently iff $r(A +_s B) = r(A) + r(B)$. Both characterizations are useful, but the latter one generalizes best to arbitrary sums, by which we mean arbitrary combinations of binary and n-ary sums, as

$$\sum A \text{ direct iff } r\left(\sum A\right) = \sum r(A)$$

To formalize this definition it would appear we need to describe arbitrary general sum expressions $\sum A$, which would require some sort of reflexion or quotation. On closer examination, however, note that we do not actually care about the exact makeup of a sum: we only need its value (a subspace), and the sum of the ranks of the summands (an integer), so we can use the type

```
Structure proper_mxsum_expr n := ProperMxsumExpr {
  proper_mxsum_val : 'M_n; proper_mxsum_rank : nat;
  _ : mxsum_spec proper_mxsum_val proper_mxsum_rank }.
```

where the inductive predicate `mxsum_spec` A s states that s is the sum of the ranks of a finite collection of matrices, whose row space sum is A. Thus, A is direct iff $r(A) = s$.

As hinted by the `Structure` keyword, we wish to declare `Canonical` instances of `proper_mxsum_expr` so that we can infer these structures from either of their two projections. This poses no problem for the proper binary and n-ary sums; however for trivial (unary) sums we would need to declare

```
Canonical Structure trivial_mxsum n A :=
  @ProperMxsum n A (\rank A) (TrivialMxsum A).
```

whose `proper_mxsum_val` projection is an arbitrary matrix A. This is interpreted by Coq as a *default* projection, which will be used eagerly for any matrix expression that is not immediately a binary or n-ary sum (the `Canonical Structure` selection process is *determinate* and driven by the head symbol of the projection value). This is undesirable because in actual use n-ary sums are often rather large expressions that need abbreviations, and we expect these to be transparent to the direct sum predicate.

Getting the right unification behavior requires a few helper structures:

```
Structure wrapped T := Wrap {unwrap : T}.
Canonical Structure wrap T x := @Wrap T x.
```

is a generic wrapper with a default instance. A unification problem `unwrap` $w \sim t$ will immediately be turned into $w \sim$ `wrap` t, unless t is of the form `unwrap` u.

We then define the `mxsum_expr` structure as a "wrapped" `proper_mxsum_expr`

```
Structure mxsum_expr m n := Mxsum {
  mxsum_val : wrapped 'M_(m, n); mxsum_rank : wrapped nat;
  _ : mxsum_spec (unwrap mxsum_val) (unwrap mxsum_rank)
}.
Canonical Structure sum_mxsum n (S : proper_mxsum_expr n) :=
  Mxsum (wrap (proper_mxsum_val S)) (wrap (proper_mxsum_rank S))
      ...
Canonical Structure trivial_mxsum m n A :=
  Mxsum (Wrap A) (Wrap (\rank A)) (TrivialMxsum A).
```

Since `wrap` is "self-inserting", matching `unwrap (mxsum_val ?)` to some arbitrary matrix expression E will first try to use `sum_mxsum`, matching T to `proper_mxsum_val ?`. This will succeed if E is a proper binary or n-ary sum; otherwise, Coq will expand `wrap` E into `Wrap` E and use `trivial_mxsum`. In effect we use the `wrapped` structure to explicitly introduce limited nondeterminism in the otherwise determinate `Canonical Structure` inference process.

With these structures we can now put

```
Definition mxdirect_def m n T
    of phantom 'M_(m, n) (unwrap (mxsum_val T)) :=
  \rank (unwrap (mxsum_val T)) == unwrap (mxsum_rank T).
Notation mxdirect A := (mxdirect_def (Phantom 'M_(_,_) A%MS)).
```

where `Inductive phantom T (x : T):= Phantom` is the generic tagged unit type. These definitions let us write `mxdirect` S for an arbitrary subspace sum S, and have Coq infer the corresponding `mxsum_expr` that actually defines the meaning of this expression. We also use the `mxsum_expr` structure to define generic lemmas about direct sum, such as

```
Lemma mxrank_sum_leqif : forall m n (S : mxsum_expr m n),
  \rank (unwrap S) <= unwrap (mxsum_rank S) ?= iff mxdirect (
      unwrap S).
```

which gives the conditionally strict rank inequality. The `leqif` predicate denoted m `<=` n `?= iff` C reads $m \leq n$, with $m = n$ iff C. The `ssrnat` library defines several combinators for `leqif`, and when applying such combinators to `mxrank_sum_leqif` the unknown S can be inferred from any one of the three arguments of `leqif`, thanks to the dual set of canonical projections of `mxsum_expr`.

5 Module and Representation Theory

Giving a full account of our development of representation theory, or of its use in the proof of the Feit-Thompson Theorem, is clearly beyond the scope of this paper. This section therefore only samples the two subjects, to illustrate how the design choices of our matrix linear algebra library fare in practice.

5.1 Group Representation

Group representations are basically morphisms from a given finite group G to some general linear group, so we adopt the design pattern introduced in [5] and define representations as a `structure` that can be inferred for specific group-to-matrices functions.

```
Definition mx_repr (G : {set gT}) n (r : gT -> ’M[R]_n) :=
    r 1%g = 1%:M
  /\ {in G &, {morph r : x y / (x * y)%g >-> x *m y}}.
Structure mx_representation G n :=
  MxRepresentation {repr_mx :> gT -> ’M_n; _ : mx_repr G repr_mx}.
```

Recall that the `%g` is the Coq overloading disambiguation operator. Note that the structure encapsulates both the morphism property, and a specific subgroup on which it holds.

Given `rG : mx_representation G n` we can define the global stabilizer of a row space U, and therefore test whether U is a G-module (i.e., stable under the action of G).

```
Definition rstabs U := [set x \in G | U *m rG x <= U]%MS.
Definition mxmodule U := G \subset rstabs U.
```

Given a G-module U, we can use the matrix bases of U to define a new representation that is the corestriction of `rG` to U, by composing `rG` with the following injection and projection:

```
Definition val_submod m : ’M_(m, \rank U) -> ’M_(m, n) :=
  mulmxr (row_base U).
Definition in_submod m : ’M_(m, n) -> ’M_(m, \rank U) :=
  mulmxr (invmx (row_ebase U) *m pid_mx (\rank U)).
```

Here `mulmxr` A is the function $B \mapsto BA$, and `row_base` U, `row_ebase` U, and `pid_mx` r are the Coq lingo for what was denoted U_R, $U_{\hat{R}}$ and 1_r in section 3.1. We also give a complementary construction for the factor representation `rG`$/U$.

Results in representation theory are alternatively formulated in terms of the representation (rarely), of modules (frequently), and sometimes of algebras. For the latter we use the encoding of section 3.4:

```
Definition enveloping_algebra_mx :=
  \matrix_(i < #|G|) mxvec (rG (enum_val i)).
```

defines the *enveloping algebra* of `rG`. Note how we use the `enum_val` function provided by the `fintype` library to effectively index the matrix rows by elements of G.

Results on modules and algebra often refer to module homomorphisms. Rather than defining a predicate testing whether a linear function f (given as a matrix) is a G-homomorphism on a given submodule U, we find it more convenient to define the largest domain on which f is a G-homomorphism:

```
Definition dom_hom_mx f : 'M_n :=
  let commGf := cent_mx_fun (enveloping_algebra_mx rG) f in
  kermx (lin1_mx (mxvec \o mulmx commGf \o lin_mul_row)).
```

and then test whether U is included in `dom_hom_mx` f, as in this definition of module isomorphism

```
CoInductive mx_iso (U V : 'M_n) : Prop := MxIso f of
  f \in unitmx & (U <= dom_hom_mx f)%MS & (U *m f :=: V)%MS.
```

Note that this definition concerns modules over the *same* representation; we need another predicate `mx_rsim` to state that different representations are similar.

5.2 Simple Modules

Many results in group module theory depend on breaking down modules into minimal or *simple* submodules. For example, Schur's lemma states that a nontrivial homomorphism between simple modules yields an isomorphism:

```
Lemma mx_Schur_iso : forall U V f,
   mxsimple U -> mxsimple V -> (U <= dom_hom_mx f)%MS ->
  (U *m f <= V)%MS -> U *m f != 0 -> mx_iso U V.
```

Unlike the `mxmodule` predicate, `mxsimple` is *non-effective*. To test whether modules are simple we need a means of testing whether polynomials are reducible, which we have not assumed. As a consequence we cannot prove constructively within Coq some obvious classical properties, such as the fact that any nontrivial module contains a simple submodule. This turns out to be only a minor nuisance, because we can still prove such facts *classically*:

```
Lemma mxsimple_exists m (U : 'M_(m, n)) : mxmodule U -> U != 0 ->
  classically (exists2 V, mxsimple V & V <= U)%MS.
```

where `classically` is a simple variation on double negation

```
Definition classically P := forall b : bool, (P -> b) -> b.
```

Whenever we are trying to prove an effective property (in `bool`), the SSReflect `without loss` tactic lets us conveniently use such results in a declarative style:

```
without loss [V simV sVU]: / exists2 V, mxsimple V & V <= U.
  exact: mxsimple_exists.
```

We prove classically the existence of module decomposition series, of splitting and closure fields, and of *socles*.

The socle of a representation is the sum of all its simple modules. Within the socle simplicity and isomorphism become decidable, so once a socle is known most constructivity issues vanish. A socle can alternatively be described as the direct sum of the *components* of the representation – the sums of isomorphic simple modules. We define a `socleType` "quasi-structure" that contains enough data to compute components, and coerces uniformly to a type that contains exactly the components.

```
Record socleType := EnumSocle {
  socle_base_enum : seq ’M[F]_n;
  _ : forall M, M \in socle_base_enum -> mxsimple M;
  _ : forall M, mxsimple M -> has (mxsimple_iso M) socle_base_enum
  }.
Definition socle_enum sG := map component_mx (socle_base_enum sG).
Inductive socle_sort sG := PackSocle W of W \in socle_enum sG.
Coercion socle_sort : socleType >-> sortClass.
```

5.3 Some Classic Results

The framework we have briefly surveyed allows us to formulate and prove all of the basic results in representation theory, including:

```
Lemma mx_Maschke :
  [char F]^’.-group G -> mx_completely_reducible 1%:M.
Theorem Clifford_component_basis : forall M, mxsimple rH M ->
  {t : nat & {x_ : sH -> ’I_t -> gT |
    forall W, let sW := (\sum_j M *m rG (x_ W j))%MS in
      [/\ forall j, x_ W j \in G, (sW :=: W)%MS & mxdirect sW]}}.
Theorem mx_JordanHolder : forall U V compU compV (m := size U),
    (last 0 U :=: last 0 V)%MS ->
  m = size V /\ (exists p : ’S_m, forall i : ’I_m,
    mx_rsim (@series_repr U i compU) (@series_repr V (p i) compV))
Lemma mx_Jacobson_density :
    mx_irreducible rG -> let E_G := enveloping_algebra_mx rG in
  (’C(’C(E_G)) <= E_G)%MS.
```

Maeshke’s theorem asserts that representations in coprime characteristic are completely reducible; this is classically equivalent but constructively slightly weaker than “semi-reducible”. Clifford’s theorem explains how an irreducible (i.e., simple) representation of G decomposes into a sum of components when restricted to some $H \lhd G$. The Jordan-Hölder theorem asserts the equivalence up to permutation of module composition series U and V (implemented as matrix sequences). In finite dimension, the Jacobson density theorem asserts that the enveloping algebra of an irreducible (i.e., simple) representation is equal to its double centraliser (in infinite dimension equality is replaced by density, hence the name). It combines with Schur’s lemma to yield the definition and construction of splitting and closure fields for groups.

The regular representation of a group G interprets G as the basis of a module on which G acts by right translation. If the scalar field of the representation is a splitting field whose characteristic does not divide the order of G, then the Wedderburn structure theorem asserts that the algebra of the regular representation R_G (known as the group ring) decomposes into a direct sum of simple subrings R_i isomorphic to matrix rings. The R_i correspond to the components of the regular representations, so we formalize this result by giving an explicit construction for the R_i given a socleType sG, and then establishing all key properties of the construction.

```
Definition Wedderburn_subring (i : sG) := <<i *m R_G>>%MS.
Lemma Wedderburn_sum : (\sum_i R_ i :=: R_G)%MS.
Lemma Wedderburn_direct : mxdirect (\sum_i R_ i)%MS.
Lemma Wedderburn_is_ring : forall i, mxring (R_ i).
Lemma Wedderburn_subring_center i : (’Z(R_ i) :=: mxvec (e_ i))%MS
Lemma rank_Wedderburn_subring i : \rank (R_ i) = (n_ i ^ 2)%N.
Lemma sum_irr_degree : (\sum_i n_ i ^ 2 = nG)%N.
```

We are now using this part of the theory as the basis for the formalization of character theory needed for the second part of the Feit-Thompson Theorem proof[4].

5.4 *P*-Stability and Extraspecial Representations

One of the early driving applications for our work on matrix linear algebra and representations was the study of p-stability, an important technical property of groups of odd order that underpins the proof of the two “deep” results on which the first part of the Feit-Thompson Theorem proof is based, the Thompson Transitivity and Uniqueness theorems[3]. The variant of p-stability we are interested in is an extension to groups G with a non-trivial p-core $O_p(G)$ of the property of “no p-element of G has a quadratic minimal polynomial in a faithful representation with a characteristic p field”, whose rather technical formulation translates in Coq as

```
Definition p_stable p G :=
  forall P A : {group gT},
     p.-group P -> ’O_p^’(G) * P <| G ->
     p.-subgroup(’N_G(P)) A -> [~: P, A, A] = 1 ->
  A / ’C_G(P) \subset ’O_p(’N_G(P) / ’C_G(P)).
Theorem odd_p_stable : forall gT p G, odd #|G| -> p_stable p G.
```

The proof of this theorem is about 300 lines long, and summarizes about 6 pages of the textbook it is drawn from[2], with some improvements (like eliminating “proof by ellipsis”).

The most challenging representation theory result in [3] was Theorem 2.5, whose 240-line proof uses representation theory to derive numerical properties of the orders of a specific class of groups (semidirect products of a cyclic group acting in a prime manner on an extraspecial p-group).

```
Theorem repr_extraspecial_prime_sdprod_cycle :
   forall p n gT (G P H : {group gT}),
   p.-group P -> extraspecial P -> P ><| H = G -> cyclic H ->
   let h := #|H| in #|P| = (p ^ n.*2.+1)%N -> coprime p h ->
   {in H^#, forall x, ’C_P[x] = ’Z(P)} ->
 [/\ (h %| p ^ n + 1) || (h %| p ^ n - 1)
  & (h != p ^ n + 1)%N ->
      forall F q (rG : mx_representation F G q),
    [char F]^’.-group G -> mx_faithful rG -> rfix_mx rG H != 0)].
```

References

1. Lang, S.: Algebra. Springer, Heidelberg (2002)
2. Gorenstein, D.: Finite groups, 2nd edn. Chelsea, New York (1980)
3. Bender, H., Glauberman, G.: Local analysis for the Odd Order Theorem. London Mathematical Society Lecture Note Series, vol. 188. Cambridge University Press, Cambridge (1994)
4. Peterfalvi, T.: Character Theory for the Odd Order Theorem. London Mathematical Society Lecture Note Series, vol. 272. Cambridge University Press, Cambridge (2000)
5. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A modular formalisation of finite group theory. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 86–101. Springer, Heidelberg (2007)
6. 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)
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. Pottier, L.: User contributions in Coq, Algebra (1999), `http://coq.inria.fr/contribs/Algebra.html`
9. Blanqui, F., Coupet-grimal, S., Delobel, W., Koprowski, A.: Color: a Coq library on rewriting and termination. In: Eighth Int. Workshop on Termination, WST (2006); to appear in MSCS
10. Rudnicki, P., Schwarzweller, C., Trybulec, A.: Commutative algebra in the Mizar system. J. Symb. Comput. 32(1), 143–169 (2001)
11. Obua, S.: Proving Bounds for Real Linear Programs in Isabelle/HOL. Theorem Proving in Higher-Order Logics, 227–244 (2005)
12. Harrison, J.: A HOL Theory of Euclidian Space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
13. Cowles, J., Gamboa, R., Baalen, J.V.: Using ACL2 Arrays to Formalize Matrix Algebra. In: ACL2 Workshop (2003)
14. Stein, J.: Documentation for the formalization of Linerar Agebra, `http://www.cs.ru.nl/~jasper/`
15. Coq development team: The Coq Proof Assistant Reference Manual, version 8.3 (2010)
16. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. INRIA Technical report, `http://hal.inria.fr/inria-00258384`
17. Sozeau, M., Oury, N.: First-Class Type Classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
18. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming 13(2), 261–293 (2003)

A Formalization of Polytime Functions*

Sylvain Heraud[1] and David Nowak[2],**

[1] INRIA Sophia Antipolis - Méditerranée, France
[2] IT Strategic Planning Group, ITRI, AIST, Japan

Abstract. We present a deep embedding of Bellantoni and Cook's syntactic characterization of polytime functions. We prove formally that it is correct and complete with respect to the original characterization by Cobham that required a bound to be proved manually. Compared to the paper proof by Bellantoni and Cook, we have been careful in making our proof fully contructive so that we obtain more precise bounding polynomials and more efficient translations between the two characterizations. Another difference is that we consider functions on bitstrings instead of functions on positive integers. This latter change is motivated by the application of our formalization in the context of formal security proofs in cryptography. Based on our core formalization, we have started developing a library of polytime functions that can be reused to build more complex ones.

Keywords: implicit computational complexity, cryptography.

1 Introduction

When formally verifying algorithms, one often proves their correctness and termination, but complexity is rarely considered. However proving correctness or termination of an algorithm that is not executable in polynomial time is of little practical use. Even at a theoretical level, it might not make much sense. For instance, in the context of security proofs one has to restrict the computational power of the adversary in the model. Indeed, an adversary with unlimited computational power could break most cryptographic schemes without actually making them insecure.

One way to take into account complexity in formal verification would be to formalize a precise execution model (e.g., Turing machines) and to explicitly count the number of steps necessary for the execution of the algorithm. Such approach would be for the least tedious and would give results depending on the particular execution model that is used whereas one is mainly interested in the complexity class independently of a particular execution model. A more convenient approach is implicit computational complexity that relates programming languages with complexity classes without relying on a particular execution model nor counting explicitly execution steps.

* Partially supported by JST-CNRS Japanese-French cooperation project on computational and symbolic proofs of security.

** Supported by Kakenhi 21500046. The work presented in this paper was carried out while this author was affiliated to Research Center for Information Security, AIST.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 119–134, 2011.

The main motivation behind our work presented in this paper is its application in the context of security proofs in cryptography for restricting the computational power of the adversary so that it is feasible. Cobham's thesis asserts that being feasible is the same as being computable in polynomial time [8]. Cryptographers follow Cobham's thesis in their security proofs by assuming that the adversary is computable in probabilistic polynomial time (PPT), i.e., executable on a Turing machine extended with a read-only random tape that has been filled with random bits, and working in (worst-case) polynomial time. Moreover the class of functions computable in polynomial time (a.k.a. polytime functions) has several natural closure properties that are convenient for programming. It is in particular closed under composition and a limited kind of recursion. Cobham uses those closure properties to characterize the polytime functions independently of any particular execution model. Indeed, although in his proof he uses a particular model of Turing machine, he claims that it is quite incidental, i.e., the particularities such as the number of tapes or the chosen instruction set have no significant effect on the proof. Even adding an instruction to erase a tape or put back the head in its initial position in a single step would not break the proof [8].

Unfortunately, the characterization of Cobham is not fully syntactic: a size bound has to be proved on the semantics of recursive functions. This thus does not allow for an automatic procedure to check whether a program satisfies or not the conditions to be in Cobham's class. About 30 years later, Bellantoni and Cook have proposed a syntactic mechanism to control the growth rate of functions and thus eliminate the need for an explicit size bound [6]. Being a fully syntactic characterization, membership in the Bellantoni-Cook's class can be checked automatically. They show that the existence of an algorithm in Cobham's class is equivalent to the existence of an algorithm in Bellantoni-Cook's class that computes the same function. This makes their class a sound and complete characterization of polytime functions: any function definable in Bellantoni-Cook's (or Cobham's) class is computable in polynomial time, and any function computable in polynomial time is definable in Bellantoni-Cook's (and Cobham's) class.

Related Work. It is not uncommon that a few months or a few years after a so-called security proof for a cryptographic scheme is published (e.g., in a top-level conference in cryptography), an attack on this same scheme is published. This shows that there is a need for formal verification in cryptography. This need is well-known among and acknowledged by cryptographers [11]. As a matter of fact, these last few years, several frameworks for machine-checking security proofs in cryptography have been proposed [3,4,15]. However, these frameworks either ignore complexity-theoretic issues or postulate the complexity of the involved functions.

Zhang has proposed a probabilistic programming language with a type system to ensure computation in probabilistic polynomial time and an equational logic to reason about those programs [22]. In [16], it has been applied to security proofs in cryptography. Zhang rely on Hofmann's SLR [12] and its extension to the probabilistic case by Mitchell et al. [14]. Those latter work are about functions on positive integers. Like us in this paper, Zhang made the move to bitstrings in order to be applicable in the context of cryptography where, for example, the bitstrings 0 and 00 are considered different although they would be identified if they were interpreted as positive integers.

In [19], it is acknowledged the need for a "polytime checker", possibly based on Bellantoni and Cook's work, and to be used to check automatically that a reduction between two NP-complete problems is computable in polynomial time. In this paper, we provide such polytime checker.

There are many other criteria to ensure that functions, defined using various programming paradigms, are in particular complexity classes. To cite only a few, some propose logical characterizations of polytime functions [13,20] or characterizations in terms of rewrite system [1]. Others deal with different complexity classes [2]. To the best of our knowledge, none of those criteria have been applied to cryptography.

Contributions. In the proof assistant Coq, we have deep embedded the bitstring versions of Cobham and Bellantoni-Cook's classes and their relation. Initially, those classes were about functions on positive integers. But in the context of cryptography we must deal with bitstrings. The reformulation of Cobham's class with bitstrings and the proof that it contains exactly the function computable in polynomial time was done in [21]. In a similar way, we have reformulated the definition of Bellantoni-Cook's class.

We have also extended Bellantoni and Cook's proof that their class is equivalent to Cobham's one by making it fully constructive, i.e., we provide explicit algorithms to perform translations between the two classes. Those algorithms can be executed in Coq and extracted automatically into a certified translator in an ML dialect supported by Coq. We also make more precise the bounding polynomials thus obtaining better bounds and a more efficient translation, whereas Bellantoni and Cook overapproximate them since they are only interested in their existence and do not try to optimize the translations.

We have started to implement libraries of functions in Cobham's and Bellantoni-Cook's classes that can be used to build more complex functions.

In the context of security proofs in cryptography, we have shown how to apply our work with the second author's toolbox for certifying cryptographic primitives [15]. We have also extended Certicrypt [4] with support to define in Bellantoni-Cook's class the mathematical functions used by adversaries: The benefit is that one gets for free polynomials that had to be postulated before our extension, thus bringing more confidence in the security proof.

We have proved a new result on Bellantoni-Cook's class: we give explicitly a polynomial that bounds the running time of a function in Bellantoni-Cook's class. Such explicit polynomial was necessary to interface our library with Certicrypt.

Outline. We start by some preliminaries in Section 2. In Section 3, we formalize Cobham's and Bellantoni-Cook's classes that characterize polytime functions. Then in Sections 4 and 5 we respectively formalize the translation from

Bellantoni-Cook's class to Cobham's class and vice versa. Finally in Section 6 we show how our formalization can be used for the purpose of formalizing security proofs in cryptography.

2 Preliminaries

In this section, we introduce our formalization of multivariate polynomials and various notations that will be used in the rest of this paper.

2.1 Multivariate Polynomials

We have implemented a library of positive multivariate polynomials. A shallow embedding of polynomials might consist in representing them as a particular class of functions on positive integers. However, since we need in Section 4 to translate polynomials into expressions in Cobham's class, we have opted for a deep embedding. A polynomial is represented as a pair of the number of distinct variables and a list of monomials. A monomial is represented as a pair of a constant positive integer and a list of variables and their powers. A variable is represented as an integer. For example, the polynomial $3y^3 + 5x^2y + 16$ is represented by $(2, [(3, [(1, 3)]); (5, [(0, 2); (1, 1)]); (16, [])])$ where the leftmost 2 is the number of variables, and variables x and y are respectively represented by 0 and 1. We chose to put the number of variables in the representation of a polynomial so as to easily inject a polynomial using m variables into the class of polynomials with n variables when $n > m$. Otherwise we would have to add artificial occurrences of variables with coefficient 0. In the library we provide utility functions in Coq to create and combine polynomials (constant, variables, addition, multiplication, composition...). We use those functions when building a polynomial. Those functions are parameterized by the number of variables, but we will omit this parameter in the rest of the paper since it will be clear from context. We write $x_0, \dots, x_{n-1}$ for the variables of a polynomial with n variables. If P is a polynomial with m variables and $\overline{Q} = \langle Q_0, \dots, Q_{m-1}\rangle$ is a vector of polynomials with n variables, we write $P(\overline{Q})$ for the polynomial with n variables defined by substituting each variable x_i in P by the polynomial Q_i and by applying distributivity of multiplication over addition and associativity of addition.

In [10], multivariate polynomials are represented in sparse Horner form and thus allow for a more efficient numerical evaluation of polynomials. Since we do not intend to numerically evaluate polynomials, we have opted for a more direct approach. This will moreover facilitate the connection with univariate polynomials in Certicrypt (cf. Section 6.2).

2.2 Notations

We list some notations that will be useful to present the results and their proofs in a concise manner. However, the meaning of those notations should be clear from the context. We write:

- xb for the concatenation of the bitstring x with a bit b in the least significant position.
- $\overline{x}$ for a vector $\langle x_0, \ldots, x_{n-1}\rangle$ (for some n).
- $\overline{x}, \overline{y}$ for the concatenation of vectors $\overline{x}$ and $\overline{y}$.
- $|\overline{x}|$ for the length of a vector $\overline{x}$;
- $\underline{|x|}$ for the size of a bitstring x;
- $|x|$ for the vector of sizes of the components of the vector $\overline{x}$, i.e., if $\overline{x} = \langle x_0, \ldots, x_{n-1}\rangle$ then $\overline{|x|} = \langle |x_0|, \ldots, |x_{n-1}|\rangle$;
- $f(x)$ for the vector of applications of f to each component of the vector $\overline{x}$, i.e., if $\overline{x} = \langle x_0, \ldots, x_{n-1}\rangle$ then $\overline{f(x)} = \langle f(x_0), \ldots, f(x_{n-1})\rangle$.
- $\overline{f}(x)$ for the vector of applications of each component of the vector $\overline{f}$ to x, i.e., if $\overline{f} = \langle f_0, \ldots, f_{n-1}\rangle$ then $\overline{f}(x) = \langle f_0(x), \ldots, f_{n-1}(x)\rangle$.

3 Characterizing Polytime Functions

In this section, we explain our deep embedding of the bitstring versions of Cobham's and Bellantoni-Cook's classes, and state some of their bounding properties.

3.1 Cobham's Class

In a seminal paper [8], Cobham characterized polytime functions as the least class of functions containing certain initial functions and closed under composition and a certain kind of recursion. However this characterization is not fully syntactic as it requires a size bound to be proved on the semantics of recursive functions.

We use the reformulation of Cobham's class taken from [21] and that deals with bitstrings instead of positive integers as it was the case in [8].

The syntax of Cobham's class $\mathcal{C}$ is given by:

$$\begin{array}{rcll}
\mathcal{C} & ::= & O & \text{constant zero} \\
& | & \Pi_i^n & \text{projection} \quad (i < n) \\
& | & S_b & \text{successor} \\
& | & \# & \text{smash} \\
& | & \mathsf{Comp}^n\ h\ \overline{g} & \text{composition} \\
& | & \mathsf{Rec}\ g\ h_0\ h_1\ j & \text{recursion}
\end{array}$$

where i and n are positive integers, b is a bit, g, h, h_0, h_1 and j are expressions in $\mathcal{C}$, and $\overline{g}$ is a vector of expressions in $\mathcal{C}$. Well-formed expressions e in $\mathcal{C}$ have a well defined arity $\mathcal{A}(e)$ given by:

$$\mathcal{A}(O) = 0 \qquad \mathcal{A}(\Pi_i^n) = n \qquad \mathcal{A}(S_b) = 1 \qquad \mathcal{A}(\#) = 2$$

$$\frac{\mathcal{A}(h) = a_h \quad |\overline{g}| = a_h \quad \forall g \in \overline{g}, \mathcal{A}(g) = n}{\mathcal{A}(\mathsf{Comp}^n\ h\ \overline{g}) = n}$$

$$\frac{\mathcal{A}(g) = a_g \quad \mathcal{A}(h_0) = \mathcal{A}(h_1) = a_h \quad \mathcal{A}(j) = a_j \quad a_h = a_g + 2 = a_j + 1}{\mathcal{A}(\mathsf{Rec}\ g\ h_0\ h_1\ j) = a_j}$$

In our implementation, $\mathcal{A}$ is a Coq function that computes the arity of a Cobham's expression if it is well formed, or returns an error message otherwise. It is helpful when programming and debugging polytime functions in Cobham's class.

The semantics is given by:

- O denotes the constant function that always returns the empty bitstring ϵ.
- $\Pi_i^n(x_0, \ldots, x_{n-1})$ is equal to x_i.
- $S_b(x)$ is equal to xb.
- $\#(x, y)$ is equal to $1\underbrace{0\ldots\ldots0}_{|x|.|y| \text{ times}}$.
- $\mathsf{Comp}^n\ h\ \overline{g}$ is equal to the function f such that $f(\overline{x}) = h(\overline{g}(\overline{x}))$.
- $\mathsf{Rec}\ g\ h_0\ h_1\ j$ is equal to the function f such that:

$$\begin{array}{ll} f(\epsilon, \overline{x}) & = g(\overline{x}) \\ f(yi, \overline{x}) & = h_i(y, f(y, \overline{x}), \overline{x}) \\ |f(y, \overline{x})| \leq |j(y, \overline{x})| & \qquad\qquad (\mathit{RecBounded}) \end{array}$$

We illustrate Cobham's class by implementing the binary successor function:

$$\begin{array}{lll} & & \mathsf{Rec} \\ \mathsf{Succ}(\epsilon) & = 1 & \quad (\mathsf{Comp}_0\ S_1\ O) \\ \mathsf{Succ}(x0) & = x1 & \quad (\mathsf{Comp}_0\ S_1\ \Pi_0^2) \\ \mathsf{Succ}(x1) & = Succ(x)0 & \quad (\mathsf{Comp}_0\ S_0\ \Pi_1^2) \\ \mathsf{Succ}(x) & \leq x1 & \quad (\mathsf{Comp}_0\ S_1\ \Pi_0^1) \end{array}$$

We prove in the following proposition that the output of a Cobham's function is bounded by a polynomial in the lengths of its inputs.

Proposition 1. *For all f in $\mathcal{C}$ with a well-defined arity $\mathcal{A}(f)$ and semantics (i.e., satisfying the condition RecBounded), there exists a length-bounding monotone polynomial $\mathsf{Pol}_{\mathcal{C}}(f)$ such that $|f(\overline{x})| \leq (\mathsf{Pol}_{\mathcal{C}}(f))(\overline{|x|})$.*

Proof. By induction on the syntax of f. Our proof is fully constructive in the sense that we define explicitly $\mathsf{Pol}_{\mathcal{C}}$. For any f in $\mathcal{C}$ with arity $\mathcal{A}(f) = n$, $\mathsf{Pol}_{\mathcal{C}}(f)$ is the monotone polynomial with n variables x_0, …, x_{n-1} defined by:

$$\begin{array}{lcl} \mathsf{Pol}_{\mathcal{C}}(O) & = & 0 \\ \mathsf{Pol}_{\mathcal{C}}(\Pi_i^n) & = & x_i \\ \mathsf{Pol}_{\mathcal{C}}(S_b) & = & x_0 + 1 \\ \mathsf{Pol}_{\mathcal{C}}(\#) & = & x_0.x_1 + 1 \\ \mathsf{Pol}_{\mathcal{C}}(\mathsf{Comp}^n\ h\ \overline{g}) & = & (\mathsf{Pol}_{\mathcal{C}}(h))(\overline{\mathsf{Pol}_{\mathcal{C}}(g)}) \\ \mathsf{Pol}_{\mathcal{C}}(\mathsf{Rec}\ g\ h_0\ h_1\ j) & = & \mathsf{Pol}_{\mathcal{C}}(j) \end{array}$$ □

We define a translation $\mathsf{Poly}{\rightarrow}\mathcal{C}$ from polynomials into Cobham's expressions. It is such that, for any polynomial P, $\mathsf{Poly}{\rightarrow}\mathcal{C}(P)$ is a unary encoding of P in Cobham's class, i.e., $|\mathsf{Poly}{\rightarrow}\mathcal{C}(P)(\overline{x})| = P(\overline{|x|})$.

3.2 Bellantoni-Cook's Class

Bellantoni and Cook have given a fully syntactic characterization of polytime functions that does not require any explicit mechanism to count the number of computation steps [6]. The control of the growth rate of functions is achieved by distinguishing two kinds of variables: the "normal" and "safe" ones written respectively on the left and right side of a semicolon such as:

$$f(\underbrace{x_0, \ldots, x_{n-1}}_{\text{normal}}; \underbrace{x_n, \ldots, x_{n+s-1}}_{\text{safe}})$$

The syntax of Bellantoni-Cook's class $\mathcal{B}$ is given by:

$$\begin{array}{rcll}
\mathcal{B} & ::= & 0 & \text{constant zero} \\
& | & \pi_i^{n,s} & \text{projection} \quad (i < n + s) \\
& | & s_b & \text{successor} \\
& | & \mathsf{pred} & \text{predecessor} \\
& | & \mathsf{cond} & \text{conditional} \\
& | & \mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S} & \text{composition} \\
& | & \mathsf{rec}\ g\ h_0\ h_1 & \text{recursion}
\end{array}$$

where i, n and s are positive integers, b is a bit, g, h, h_0 and h_1 are expressions in $\mathcal{B}$, and $\overline{g_N}$ and $\overline{g_S}$ are vectors of expressions in $\mathcal{B}$. Note that, contrary to Cobham's class, a bounding function j is not needed for recursion. Well-formed expressions e in $\mathcal{B}$ have well defined arities $\mathcal{A}(e)$ (counting separately the numbers of normal and safe variables) given by:

$$\mathcal{A}(0) = (0, 0) \qquad \mathcal{A}(\pi_i^{n,s}) = (n, s) \qquad \mathcal{A}(s_b) = (0, 1)$$
$$\mathcal{A}(\mathsf{pred}) = (0, 1) \qquad \mathcal{A}(\mathsf{cond}) = (0, 4)$$

$$\frac{\begin{array}{c}\mathcal{A}(h) = (n_h, s_h) \quad |\overline{g_N}| = n_h \quad |\overline{g_S}| = s_h \\ \forall g \in \overline{g_N}, \mathcal{A}(g) = (n, 0) \quad \forall g \in \overline{g_S}, \mathcal{A}(g) = (n, s)\end{array}}{\mathcal{A}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) = (n, s)}$$

$$\frac{\mathcal{A}(g) = (n_g, s_g) \quad \mathcal{A}(h_0) = \mathcal{A}(h_1) = (n_h, s_h) \quad n_h = n_g + 1 \quad s_h = s_g + 1}{\mathcal{A}(\mathsf{rec}\ g\ h_0\ h_1) = (n_h, s_g)}$$

This function $\mathcal{A}$ is implemented like the one for Cobham's class.

The semantics is given by:

- 0 denotes the constant function that always returns the empty bitstring ϵ.
- $\pi_i^{n,s}(x_0, \ldots, x_{n-1}; x_n, \ldots, x_{n+s-1})$ is equal to x_i.
- $s_b(; x)$ is equal to xb.
- $\mathsf{pred}(; \epsilon) = \epsilon$ and $\mathsf{pred}(; xi) = x$.
- $\mathsf{cond}(; \epsilon, x, y, z) = x$, $\mathsf{cond}(; w0, x, y, z) = y$ and $\mathsf{cond}(; w1, x, y, z) = z$.
- $\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}$ is equal to the function f such that:

$$f(\overline{x}; \overline{y}) = h(\overline{g_N}(\overline{x};); \overline{g_S}(\overline{x}; \overline{y}))$$

Note here that the functions in $\overline{g_N}$ only have access to normal variables.

– $\mathsf{rec}\ g\ h_0\ h_1$ is equal to the function f such that:

$$\begin{aligned} f(\epsilon, \overline{x}; \overline{y}) &= g(\overline{x}; \overline{y}) \\ f(zi, \overline{x}; \overline{y}) &= h_i(z, \overline{x}; f(z, \overline{x}; \overline{y}), \overline{y}) \end{aligned}$$

Note here that the result of the recursive call $f(z, \overline{x}; \overline{y})$ is passed at a safe position. This prevents it to be used as the recursion argument in a nested recursion.

One can see that, contrary to Cobham's class $\mathcal{C}$, there is no size bound to be proved on recursive functions: Bellantoni-Cook's class $\mathcal{B}$ is syntactically defined.

Reader may have noticed that our definition of Bellantoni-Cook's class is slightly different from the one in [6]. First, here the conditional cond distinguishes between three cases (empty, even or odd bitstrings), whereas in [6] the empty bitstring is treated as an even one. Second, here the base case for recursion is the empty bitstring, whereas in [6] it is any bitstring whose interpretation as a positive integer is 0, i.e., the empty bitstring or any bitstring made of any number of bits 0 only. We made those changes because in cryptography one wants to distinguish, for example, bitstrings 0 and 00 although they would have the same interpretation in terms of positive integers. These changes are validated by the results we proved in the rest of the paper where we translate our Bellantoni-Cook's expressions to/from the bitstring version of Cobham's expressions [21].

The following examples illustrate how one can program addition and multiplication in Bellantoni-Cook's class and their respective arity:

$$\begin{array}{ll} \mathit{plus} := \mathsf{rec} & \qquad\qquad \mathit{mult} := \mathsf{rec} \\ \quad (\pi_0^{0,1}) & \qquad\qquad \quad (\mathsf{comp}^{1,0}\ O\ \langle\rangle\ \langle\rangle) \\ \quad (\mathsf{comp}^{1,2}\ S_1\ \langle\rangle\ \langle\pi_1^{1,2}\rangle) & \qquad\qquad \quad (\mathsf{comp}^{1,2}\ \mathit{plus}\ \langle\pi_1^{2,0}\rangle\ \langle\pi_2^{2,1}\rangle) \\ \quad (\mathsf{comp}^{1,2}\ S_1\ \langle\rangle\ \langle\pi_1^{1,2}\rangle) & \qquad\qquad \quad (\mathsf{comp}^{1,2}\ \mathit{plus}\ \langle\pi_1^{2,0}\rangle\ \langle\pi_2^{2,1}\rangle) \\ & \\ \mathcal{A}(\mathit{plus}) = (1,1) & \qquad\qquad \mathcal{A}(\mathit{mult}) = (2,0) \end{array}$$

We prove in the following proposition that the output of a Bellantoni-Cook's function is bounded by the sum of a polynomial in the lengths of its normal inputs and the size of its longest safe input. This is so because syntactic restrictions ensure that we cannot increase the lengths of safe inputs by more than an additive constant that will be taken into account in the polynomial part.

Proposition 2 (Polymax Bounding). *For all f in $\mathcal{B}$ with well-defined arities $\mathcal{A}(f)$, there exists a length-bounding monotone polynomial $\mathsf{Pol}_{\mathcal{B}}(f)$ such that, for all $\overline{x}$ and $\overline{y}$, $|f(\overline{x}; \overline{y})| \leq (\mathsf{Pol}_{\mathcal{B}}(f))(\overline{|x|}) + \mathsf{max}_i|y_i|$.*

Proof. By induction on the syntax of f. Our proof is fully constructive in the sense that we define explicitly $\mathsf{Pol}_{\mathcal{B}}$. For any f in $\mathcal{B}$ with arity $\mathcal{A}(f) = (n, s)$, $\mathsf{Pol}_{\mathcal{B}}(f)$ is the monotone polynomial with n variables $x_0, \ldots, x_{n-1}$ defined by:

$$
\begin{array}{lcl}
\mathsf{Pol}_{\mathcal{B}}(0) & = & 0 \\
\mathsf{Pol}_{\mathcal{B}}(\pi_i^{n,s}) & = & x_i \text{ if } i < n \\
 & & 0 \ \text{ otherwise} \\
\mathsf{Pol}_{\mathcal{B}}(s_b) & = & 1 \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{pred}) & = & 0 \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{cond}) & = & 0 \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) & = & \mathsf{Pol}_{\mathcal{B}}(h)(\overline{\mathsf{Pol}_{\mathcal{B}}(g_N)}) + \sum(\overline{\mathsf{Pol}_{\mathcal{B}}(g_S)}) \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1) & = & \mathsf{shift}(\mathsf{Pol}_{\mathcal{B}}(g)) + x_0.(\mathsf{Pol}_{\mathcal{B}}(h_0) + \mathsf{Pol}_{\mathcal{B}}(h_1))
\end{array}
$$

where $\mathsf{shift}(P)$ is the polynomial P with each variable x_i replaced by x_{i+1}. □

We define a translation $\mathsf{Poly} \rightarrow \mathcal{B}$ from polynomials into Bellantoni-Cook's expressions. It is such that, for any polynomial P, $\mathsf{Poly}\rightarrow\mathcal{B}(P)$ is a unary encoding of P in Bellantoni-Cook's class, i.e., $|\mathsf{Poly}\rightarrow\mathcal{B}(P)(\overline{x})| = P(\overline{|x|})$.

In order to ease further development of functions in Bellantoni-Cook's class, we provide a mechanism to infer automatically the optimal values for the parameters n and s appearing in $\pi_i^{n,s}$ and $\mathsf{comp}^{n,s}$ thus obtaining more elegant code. This is implemented with a new syntax $\mathcal{B}_{\mathsf{inf}}$ for Bellantoni-Cook's class where arities do not appear in the syntax. We have validated this new syntax by providing certified translations between $\mathcal{B}_{\mathsf{inf}}$ and $\mathcal{B}$. When translating from $\mathcal{B}_{\mathsf{inf}}$ to $\mathcal{B}$, we can force arities to be larger that the minimal ones that are inferred.

4 Compiling Bellantoni-Cook into Cobham

In this section, we provide our formalization of the translation of expressions in Bellantoni-Cook's class into expressions in Cobham's class. The main result is stated in the following theorem.

Theorem 1. *For all f in $\mathcal{B}$ with well defined arities $\mathcal{A}(f)$, there exists f' in $\mathcal{C}$ such that for all vectors of bitstrings $\overline{x}$ and $\overline{y}$, $f(\overline{x};\overline{y}) = f'(\overline{x},\overline{y})$.*

Proof. The proof is split into two inductions on the syntax of f: The first one to prove the equality and the second one to prove that the condition *RecBounded* is satisfied. Our proof is fully constructive in the sense that we define explicitly the translation $\mathcal{B} \rightarrow \mathcal{C}$ from $\mathcal{B}$ to $\mathcal{C}$ and define f' as $\mathcal{B}\rightarrow\mathcal{C}(f)$. The difficulty of the proof is in the generation of a Cobham expression that satisfies the condition *RecBounded*, and to build the polynomial j that bounds the recursive calls.

- The first cases are immediate: $\mathcal{B}\rightarrow\mathcal{C}(0) \ = \ O$
  $\mathcal{B}\rightarrow\mathcal{C}(\pi_i^{n,s}) \ = \ \Pi_i^{n+s}$
  $\mathcal{B}\rightarrow\mathcal{C}(s_b) \ = \ S_b$
- pred and cond are translated by using Rec:

$$
\begin{array}{lcl}
\mathcal{B}\rightarrow\mathcal{C}(\mathsf{pred}) & = & \mathsf{Rec}\ O\ \Pi_0^2\ \Pi_0^2\ \Pi_0^1 \\
\mathcal{B}\rightarrow\mathcal{C}(\mathsf{cond}) & = & \mathsf{Rec}\ \Pi_0^3\ \Pi_4^5\ \Pi_3^5 \\
 & & \quad \mathsf{Comp}^4\ \#\ \langle \\
 & & \qquad \mathsf{Comp}^4\ S_1\ \langle \Pi_1^4 \rangle; \\
 & & \qquad \mathsf{Comp}^4\ \#\ \langle\ \mathsf{Comp}^4\ S_1\ \langle \Pi_2^4 \rangle; \mathsf{Comp}^4\ S_1\ \langle \Pi_3^4 \rangle\ \rangle\rangle
\end{array}
$$

- For $\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}$ we need to add dummy variables, since the functions in $\overline{g_N}$ do not take the safe arguments as parameters. We need to transform these functions in $\overline{g_N}$ into functions with arity $n+s$. $\mathsf{dummies}_s$ (written in $\mathcal{C}$) add s dummy variables that are ignored:

$$\mathcal{B}\!\rightarrow\!\mathcal{C}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) \;=\; \mathsf{Comp}^{n+s}\quad \mathcal{B}\!\rightarrow\!\mathcal{C}(h) \left(\overline{\mathsf{dummies}_s\ (\mathcal{B}\!\rightarrow\!\mathcal{C}(g_N))}, \overline{\mathcal{B}\!\rightarrow\!\mathcal{C}(g_S)}\right)$$

- For $\mathsf{rec}\ g\ h_0\ h_1$ we need to change the order of the arguments passed to the translations of h_0 and h_1. Indeed, while in $\mathcal{B}$ the recursive argument is put after the normal ones, in $\mathcal{C}$ it should be the second argument. This reordering is done by the function $\mathsf{move\_arg}_{2,n}$ (written in $\mathcal{C}$). Moreover, we need to derive a suitable bound for the fourth argument of Rec. By Proposition 2 and the fact that the sum $|x_n| + \cdots + |x_{n+s-1}|$ of the sizes of the safe arguments is greater than or equal to the maximum size of the safe arguments, we can take the polynomial $\mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1) + x_n + \cdots + x_{n+s-1}$ for the bound. We then use $\mathsf{Poly}\!\rightarrow\!\mathcal{C}$ to encode it in $\mathcal{C}$:

$$\begin{aligned}\mathcal{B}\!\rightarrow\!\mathcal{C}(\mathsf{rec}\ g\ h_0\ h_1) \;=\; & \mathsf{Rec} \\ & \quad \mathcal{B}\!\rightarrow\!\mathcal{C}(g) \\ & \quad \mathsf{move\_arg}_{2,n}\ (\mathcal{B}\!\rightarrow\!\mathcal{C}(h_0)) \\ & \quad \mathsf{move\_arg}_{2,n}\ (\mathcal{B}\!\rightarrow\!\mathcal{C}(h_1)) \\ & \quad \mathsf{Poly}\!\rightarrow\!\mathcal{C}\begin{pmatrix}\mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1)\ + \\ x_n\ +\ \cdots\ +\ x_{n+s-1}\end{pmatrix}\end{aligned} \qquad \square$$

5 Compiling Cobham into Bellantoni-Cook

Contrary to Bellantoni-Cook's class $\mathcal{B}$, one does not distinguish between normal and safe variables in Cobham's class $\mathcal{C}$. In $\mathcal{C}$ it is possible to recur on any argument, whereas in $\mathcal{B}$ one can only recur on normal arguments. Thus, when translating from $\mathcal{C}$ to $\mathcal{B}$, we must introduce a distinction and deal appropriately with recursion. In our formalization, we follow Bellantoni and Cook's translation scheme by assuming that all the arguments are safe and adding an artificial normal argument w whose length is large enough to ensure enough recursion steps. This gives us the lemma below. After that, we will get rid of w thus obtaining Theorem 2.

Lemma 1 (Recursion Simulation). *For all function f in $\mathcal{C}$ with well-defined arity $\mathcal{A}(f) = n$ and semantics (i.e., satisfying the condition RecBounded), there exists an f' in $\mathcal{B}$ and a monotone polynomial $\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)$ such that for all vector of bitstrings $\overline{x}$ and bistring w such that $\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)(\overline{|x|}) \leq |w|$, $f(\overline{x}) = f'(w; \overline{x})$.*

Proof. By induction on the syntax of f. Our proof is fully constructive in the sense that we define explicitly the polynomial $\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)$ and the translation $\mathcal{C}\!\rightarrow\!\mathcal{B}$, and define f' as $\mathcal{C}\!\rightarrow\!\mathcal{B}(f)$. Our translation is such that if $\mathcal{A}(f) = n$ then $\mathcal{A}(f') = (1, n)$, i.e., f' takes one normal argument and n safe arguments.

- The first cases for $\mathcal{C}\to\mathcal{B}(f)$ are immediate. We just have to make sure that the arities are right:

$$\begin{array}{lcl}
\mathcal{C}\to\mathcal{B}(O) & = & \mathsf{comp}^{1,n}\ O\ \langle\rangle\ \langle\rangle \\
\mathcal{C}\to\mathcal{B}(\Pi^n_i) & = & \pi^{1,n}_{i+1} \\
\mathcal{C}\to\mathcal{B}(S_b) & = & \mathsf{comp}^{1,n}\ s_b\ \langle\rangle\ \langle\pi^{1,n}_1\rangle \\
\mathcal{C}\to\mathcal{B}(\mathsf{Comp}^n\ h\ \overline{g}) & = & \mathsf{comp}^{1,n}\ (\mathcal{C}\to\mathcal{B}(h))\ \langle\pi^{1,0}_1\rangle\ \overline{\mathcal{C}\to\mathcal{B}(g)}
\end{array}$$

$\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)(\overline{|x|})$ is also immediate for these first cases:

$$\begin{array}{lcl}
\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(O) & = & 0 \\
\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(\Pi^n_i) & = & 0 \\
\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(S_b) & = & 0 \\
\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(\mathsf{Comp}^n\ h\ \overline{g}) & = & \mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(h)(\overline{\mathsf{Pol}_{\mathcal{C}}(g)}) + \sum\limits_{g\in\overline{g}} \mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(g)
\end{array}$$

  The rightness of the case for Comp follows by induction hypothesis and Proposition 1.

- For the case of $\mathsf{Rec}\ g\ h_0\ h_1\ j$, we follow [6] in defining intermediate functions in $\mathcal{B}$. However we need less of them since our definition of $\hat{f}$ below is simpler. We define P in $\mathcal{B}$ such that $P(a;b)$ removes the $|a|$ least significant bits of b, i.e., $P(\epsilon;b)=b$ and $P(ai;b)=\mathsf{pred}(;P(a;b))$. We define Y in $\mathcal{B}$ such that $Y(z,w;y)$ removes the $|w|-|z|$ least significant bits of y, i.e., $Y(z,w;y)=P(P'(z,w);y)$ where $P'(a,b;)=P(a;b)$. P and Y are then used to define $\hat{f}$ in $\mathcal{B}$:

$$\begin{array}{ll}
\hat{f}(\epsilon,w;y,\overline{x}) & = g(w;\overline{x}) \\
\hat{f}(zj,w;y,\overline{x}) = & \begin{cases} g'(w;\overline{x}) & \text{if } Y(S_1z,w;y) \text{ is } \epsilon \\ h'_0(w,Y(z,w;y),\hat{f}(z,w;y,\overline{x}),\overline{x}) & \text{if } Y(S_1z,w;y) \text{ is even} \\ h'_1(w;Y(z,w;y),\hat{f}(z,w;y,\overline{x}),\overline{x}) & \text{if } Y(S_1z,w;y) \text{ is odd} \end{cases}
\end{array}$$

  where g', h'_0 and h'_1 are respectively $\mathcal{C}\to\mathcal{B}(g)$, $\mathcal{C}\to\mathcal{B}(h_0)$ and $\mathcal{C}\to\mathcal{B}(h_1)$. Our definition of $\hat{f}$ is simpler than in [6] because we do not need, like in [6], an additional intermediate function to check whether y is an encoding of the positive integer 0. In our case, we stop the recursion when y is equal to ϵ, since the cond can check whether the first safe argument is ϵ.

  We then define $f'(w;y,\overline{x})$ in $\mathcal{B}$ such that it is equal to $\hat{f}(w,w;y,\overline{x})$, and finally:

$$\mathcal{C}\to\mathcal{B}(\mathsf{Rec}\ g\ h_0\ h_1\ j) = f'\ (\mathcal{C}\to\mathcal{B}(g))\ (\mathcal{C}\to\mathcal{B}(h_0))\ (\mathcal{C}\to\mathcal{B}(h_1))$$

  For the polynomial, we have:

$$\begin{array}{l}
\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(\mathsf{Rec}\ g\ h_0\ h_1\ j)(|y|,\overline{|x|}) = (\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(h_0)+\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(h_1))(|y|,\mathsf{Pol}_{\mathcal{C}}(f),\overline{|x|})+ \\
\qquad\qquad\qquad\qquad \mathsf{shift}(\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(g)(\overline{|x|})) + |y| + 2
\end{array}$$

- We can define the smash function # from $\mathcal{C}$ in $\mathcal{B}$ by a double recursion:

$$\begin{array}{lcll} \#'(\epsilon, y) & = & y & \\ \#'(xi, y) & = & \#'(x, y)\,0 & \text{(concatenation with a bit 0)} \\ \#(\epsilon, y) & = & 1 & \\ \#(xi, y) & = & \#'(y, \#(x, y)) & \end{array}$$

  In order to implement in $\mathcal{B}$ those two recursive functions we apply the same technique as in the case of Rec above. We first obtain a $\#'$ in $\mathcal{B}$ by constructing f' with $g = \pi_1^{1,1}$ and $h_0 = h_1 = \mathsf{comp}^{1,3}\ S_0\ \langle\rangle\ \langle\pi_2^{1,3}\rangle$. We then obtain # in $\mathcal{B}$ by applying the same construction of f' with $g = \mathsf{one}^{1,1}$ and $h_0 = h_1 = \mathsf{dummies}_{0,1}(\mathsf{comp}^{1,2}\ \#'\ \langle\pi_0^{1,0}\rangle\ \langle\pi_2^{1,2}; \pi_1^{1,2}\rangle)$.
  For the polynomial, we obtain (after simplification):

$$\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(\#) = x_0 + 2x_1 + 18 \qquad \square$$

Finally, the main result of this section is stated in the following theorem.

Theorem 2. *For all f in $\mathcal{C}$ with well-defined arity $\mathcal{A}(f)$ and semantics (i.e., satisfying the condition RecBounded), there exists f' in $\mathcal{B}$ such that, for all vectors of bitstrings $\overline{x}$, $f(\overline{x}) = f'(\overline{x};)$.*

Proof. We define the expression b_f in $\mathcal{B}$ by $\mathsf{Poly}{\to}\mathcal{B}(\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f))$. By definition of $\mathsf{Poly}{\to}\mathcal{B}$, for all vector of bistrings $\overline{x}$, $|b_f(\overline{x};)| = \mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)(\overline{|x|})$. We can thus apply Lemma 1 which gives $f'(\overline{x};) = (\mathcal{C} \to \mathcal{B}(f))(b_f(\overline{x};); \overline{x})$. □

Our translation gives a more efficient code than the one in [6] since our definition of b_f is more precise: the number of recursive calls will be no more than what is strictly necessary. Indeed, authors of [6] use general properties of multivariate polynomials to first prove the existence of positive integers a and c such that $\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)(\overline{|x|}) \leq (\sum_j |x_j|)^a + c$ and then use a and c to build a b_f that satisfies the condition of Lemma 1, i.e., $\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)(\overline{|x|}) \leq |b(\overline{x};)|$. Their b_f is an overapproximation of $\mathsf{Pol}_{\mathcal{C}\to\mathcal{B}}(f)$ while our b_f is an exact encoding.

6 Applications

Security properties in cryptography are often modeled as games, and then security proofs consist in showing that no adversary can win the game [7,18]. Most of those proofs are based on computational assumptions that state that an effective adversary cannot solve a particular mathematical problem, e.g., Diffie-Hellman problems [9]. Effective adversaries are modeled as strict probabilistic polynomial-time functions, i.e., independently of the random choices, the execution time is bounded by a polynomial in a security parameter (typically the length of the inputs). This means that an adversary can be modeled as a polytime function with, as an additional parameter, a long enough bitstring that will be used by the adversary as its source of random bits.

6.1 Application to the Second Author's Toolbox

The second author's toolbox is a collection of definitions and lemmas to be used for verifying game transformations in security proofs [15]. With this toolbox, our library can be used as such when applying a computational hypothesis. The computational hypotheses can indeed be restricted to adversaries defined in Cobham's or Bellantoni-Cook's class (it is not too restrictive since those classes are complete) and adversaries appearing in proofs must then be defined in one of those classes. For example, when applying the Decisional Diffie-Hellman assumption (DDH) in the security proof for Hashed ElGamal in [15], a new adversary φ is built from two adversaries A_1 and A_2:

$$\varphi(X, Y, Z) \; =_{def} \; A_2(r, (X, k), (Y, H_k(Z) \oplus \pi_b(A_1(r, (X, k))))) \overset{?}{=} b$$

where b, k and r are fixed, $\oplus$ is a the bitwise exclusive or (xor), π_b is the b^{th} projection (b is equal to 1 or 2), and $\overset{?}{=}$ is the equality test. That A_1 and A_2 are polytime is given by hypothesis. We can also assume that the hash function H_k is polytime. Being polytime, they are definable in Bellantoni-Cook's class. Moreover, projections, exclusive or and equality test are easily defined in Bellantoni-Cook's class. Therefore φ is easily definable in Bellantoni-Cook's class and thus it is polytime.

6.2 Application to Certicrypt

The application of our library to Certicrypt requires more work but brings noticeable benefits.

In Certicrypt, a game is a probabilistic imperative program that transforms a distribution of input states into a distribution of output states. A state includes a time index. A distribution of states is polynomially bounded if there are two (univariate) polynomials p and q respectively bounding the size of the data and the time index of each state in the distribution. A program is strict probabilistic polynomial time (PPT) iff: it always terminates; and, there exists two (univariate) polynomial transformers F and G such that, for every polynomially bounded (by p and q) distribution of input states, the distribution of output states is bounded by $F(p)$ (bounding the output size) and $q + G(p)$ (bounding the execution time). Interested reader should refer to [4] for further explanation about this way to formalize PPT.

We have built an interface with Certicrypt made of the following components:

- The core language of Certicrypt can be extended with user-defined types and functions. But the time cost of each function has to be axiomatized in the current implementation of Certicrypt. We have added the possibility to include functions that have been defined in our implementation of Bellantoni-Cook's class and that are thus automatically proved executable in polynomial time, thus removing the need for postulates.

- We have added a conversion from any multivariate polynomial P given by our library into a univariate one $\lceil P \rceil$ in Certicrypt that overapproximates P when applied to the maximal argument: This is easily done by substituting all variables x_0, $\dots$, x_{n-1} in P by a single variable x: $\lceil P \rceil \ =_{def} \ P[x_0 \mapsto x; \dots; x_{n-1} \mapsto x]$.
- In the case of a program c defined in Bellantoni-Cook's class, we can take $F(p)$ to be equal to $1 \ + \ 2 \lceil \mathsf{Pol}_{\mathcal{B}}(c) \rceil (p)$. This is justified by Proposition 2. The multiplication by 2 and and addition of 1 are here because of technical reasons coming from Certicrypt. For example, the multiplication by 2 comes from the fact that the size of a boolean in Certicrypt is 2.
- In order to obtain $G(p)$, we need to consider the obvious implementation of Bellantoni-Cook's class on a stack machine as described in Section 3.4.2 of [5]. We have equipped the semantics of Bellantoni-Cook's class with a time index that keeps track of the running time. We can then prove that the multivariate polynomial $\mathsf{Pol}_{\mathsf{time}}$ below is an upper bound of the running time, and use it to define $G(p)$.

$$\mathsf{Pol}_{\mathsf{time}}(0) = \mathsf{Pol}_{\mathsf{time}}(\pi_i^{n,s}) = \mathsf{Pol}_{\mathsf{time}}(s_b) = \mathsf{Pol}_{\mathsf{time}}(\mathsf{pred}) = \mathsf{Pol}_{\mathsf{time}}(\mathsf{cond}) = 1$$

$$\begin{aligned} \mathsf{Pol}_{\mathsf{time}}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) &= \mathsf{Pol}_{\mathsf{time}}(h)(\overline{\mathsf{Pol}_{\mathcal{B}}(g_N)}) + \\ &\quad \textstyle\sum(\overline{\mathsf{Pol}_{\mathsf{time}}(g_N)}) + \sum(\overline{\mathsf{Pol}_{\mathsf{time}}(g_S)}) \\ \mathsf{Pol}_{\mathsf{time}}(\mathsf{rec}\ g\ h_0\ h_1) &= \mathsf{shift}(\mathsf{Pol}_{\mathsf{time}}(g)) + \\ &\quad x_0.(\mathsf{Pol}_{\mathsf{time}}(h_0) + \mathsf{Pol}_{\mathsf{time}}(h_1)) \end{aligned}$$

where $\mathsf{shift}(P)$ is the polynomial P with each variable x_i replaced by x_{i+1}. An interesting thing about $\mathsf{Pol}_{\mathsf{time}}$ is that, in the case of $\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}$, it is necessary to consider the size $\overline{\mathsf{Pol}_{\mathcal{B}}(g_N)}$ of the outputs of the functions in $\overline{g_N}$ for the running time of h, but not the size of the outputs of the functions in $\overline{g_S}$. This is so because syntactic restrictions ensure that we cannot increase the lengths of safe inputs by more than an additive constant. Finally, for a program c defined in Bellantoni-Cook's class, we take $G(p)$ to be equal to:

$$\lceil \mathsf{Pol}_{\mathsf{time}}(c) \rceil (p)$$

The reader might be surprised that in this section we consider a particular implementation of Bellantoni-Cook's class while in introduction we said that we are interested in complexity independently of any execution model. The reason is that, although being in Cobham's or Bellantoni-Cook's class guarantees that there exists a polynomial bounding the execution time, it does not give any clue on the actual value of this polynomial. However Certicrypt requires that we explicitly give such a polynomial. This is why we consider here a particular execution model: to be able to compute a polynomial.

7 Conclusions and Future Work

We have formalized Cobham's and Bellantoni-Cook's classes and their relations in the proof assistant Coq. Usage of proof assistant led us to formalize parts of the proofs that were only informal in Bellantoni and Cook's paper. Our formalization allows to use those classes as programming languages to define any function that is computable in polynomial time. We have shown in particular that it can be used to build adversaries in security proofs in cryptography.

Future Work. In order to facilitate the use of our formalization, an important future work is to carry on developing a convenient library of polytime functions on bitstrings that can be reused in the construction of more advanced polytime functions. It is easy to implement bitwise operations such as bitwise XOR, NOT, AND, etc. However, when implementing numerical operations such as bitwise addition, dealing with the carry bit does not fit immediately in Bellantoni-Cook's recursion scheme. One possible solution is to implement binary addition, multiplication and other numerical functions in Cobham's class such as in [17], and use our automatic translation $\mathcal{C} \to \mathcal{B}$ (defined in Section 5) to derive their implementations in Bellantoni-Cook's class. Also, our approach can be extended with higher order so as to formalize a more powerful programming language such as CSLR [22].

Acknowledgments. We are grateful to Benjamin Grégoire and Santiago Zanella Béguelin for replying so quickly to every question we have had on Certicrypt, and for their comments on an earlier draft of this paper. We thank anonymous reviewers for their helpful suggestions, all of which improved the paper.

References

1. Avanzini, M., Moser, G.: Complexity Analysis by Rewriting. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 130–146. Springer, Heidelberg (2008)
2. Arai, T., Eguchi, N.: A new function algebra of EXPTIME functions by safe nested recursion. In: ACM Transactions on Computational Logic, vol. 10(4). ACM, New York (2009)
3. Backes, M., Berg, M., Unruh, D.: A formal language for cryptographic pseudocode. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 353–376. Springer, Heidelberg (2008)
4. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages (POPL 2009), pp. 90–101. ACM, New York (2009)
5. Bellantoni, S.: Predicative Recursion and Computational Complexity. PhD Thesis, University of Toronto (1992)
6. Bellantoni, S., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity 2, 97–110 (1992)
7. 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)
8. Cobham, A.: The intrinsic computational difficulty of functions. In: Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science, pp. 24–30. North-Holland, Amsterdam (1964)

9. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory IT 22(6), 644–654 (1976)
10. 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)
11. Halevi, S.: A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181 (2005)
12. Hofmann, M.: Safe recursion with higher types and BCK-algebra. Annals of Pure and Applied Logic 104(1-3), 113–166 (2000)
13. Leivant, D.: A foundational delineation of computational feasibility. In: Sixth Annual IEEE Symposium on Logic in Computer Science, pp. 2–11. IEEE Computer Society, Los Alamitos (1991)
14. Mitchell, J.C., Mitchell, M., Scedrov, A.: A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS 1998), pp. 725–733. IEEE Computer Society, Los Alamitos (1998)
15. Nowak, D.: A framework for game-based security proofs. In: Qing, S., Imai, H., Wang, G. (eds.) ICICS 2007. LNCS, vol. 4861, pp. 319–333. Springer, Heidelberg (2007)
16. Nowak, D., Zhang, Y.: A calculus for game-based security proofs. In: Heng, S.-H., Kurosawa, K. (eds.) ProvSec 2010. LNCS, vol. 6402, pp. 35–52. Springer, Heidelberg (2010)
17. Rose, H.E.: Subrecursion: functions and hierarchies. Oxford Logic Guides 9. Clarendon Press, Oxford (1984)
18. Shoup, V.: Sequences of games: A tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004)
19. Schürmann, C., Shah, J.: Representing reductions of NP-complete problems in logical frameworks: A case study. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding (MERLIN 2003). ACM, New York (2003)
20. Schürmann, C., Shah, J.: Identifying Polynomial-Time Recursive Functions. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 525–540. Springer, Heidelberg (2005)
21. Tourlakis, G.J.: Computability, Reston (1984)
22. Zhang, Y.: The computational SLR: a logic for reasoning about computational indistinguishability. In: Mathematical Structures in Computer Science, vol. 20, pp. 951–975. Cambridge University Press, Cambridge (2010)

Three Chapters of Measure Theory in Isabelle/HOL

Johannes Hölzl* and Armin Heller

Technische Universität München
Institut für Informatik, Boltzmannstr. 3, 85748 Garching bei München, Germany
http://home.in.tum.de/~hoelzl/

Abstract. Currently published HOL formalizations of measure theory concentrate on the Lebesgue integral and they are restricted to real-valued measures. We lift this restriction by introducing the extended real numbers. We define the Borel σ-algebra for an arbitrary type forming a topological space. Then, we introduce measure spaces with extended real numbers as measure values. After defining the Lebesgue integral and verifying its linearity and monotone convergence property, we prove the Radon-Nikodým theorem (which shows the maturity of our framework). Moreover, we formalize product measures and prove Fubini's theorem. We define the Lebesgue measure using the gauge integral available in Isabelle's multivariate analysis. Finally, we relate both integrals and equate the integral on Euclidean spaces with iterated integrals. This work covers most of the first three chapters of Bauer's measure theory textbook.

1 Introduction

Measure theory plays an important role in modeling the physical world, and in particular is the foundation of probability theory. Current HOL formalizations of measure theory mostly concentrate on the Lebesgue integral [2, 10, 11]. We extend this by a number of fundamental concepts:

Lebesgue measure. To use the Lebesgue integral for functions on a real domain we need to introduce a measure on $\mathbb{R}$. The Lebesgue measure $\boldsymbol{\lambda}$ assigns the length $b - a$ to every interval $[a, b]$, and is closed under countable union and difference. The Lebesgue integral on $\boldsymbol{\lambda}$ is an extension of the Riemann integral.

Product measure. Defines a measure on the product of two or more measure spaces. We can also represent Euclidean spaces as products of the Lebesgue measure on $\mathbb{R}$. This is also necessary to prove Fubini's theorem, i.e., the commutativity of integrals.

Extended real numbers. The introduction of Lebesgue measure requires infinite measure values, hence we introduce the extended real numbers and use them as measure values.

* Supported by the DFG Graduiertenkolleg 1480 (PUMA).

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 135–151, 2011.
Springer-Verlag Berlin Heidelberg 2011

Radon-Nikodým derivative. Given two measures ν and μ, we can represent ν with density f with respect to μ, under certain assumptions.

$$\nu\ A = \int_A f\ d\mu$$

This density f is called the Radon-Nikodým derivative. The existence of such a density is used in information theory to define *mutual information* and in probability theory to define *conditional expectation*.

Restricted forms of these concepts where already formalized in HOL theorem provers [6, 2, 11, 10, 8, 5]. By formalizing these concepts in a more generic way, it is now possible to combine the results of these works. With the Lebesgue measure, the Radon-Nikodým theorem and the product measure we formalize most of the first three chapters (∼ 70 pages) of Bauer's textbook about measure theory [1]. We only show the theorem statements, but the full proofs are publicly available in the current development version of Isabelle.[1]

2 Preliminaries

We use the following concepts and notations: We write the power set as $\mathcal{P}(A) = \{B \mid B \subseteq A\}$, the universe for type α as $\mathcal{U} :: \alpha\, set$, $\bot :: \alpha$ is an arbitrary element of type α, the image of A under f is $f[A] = \{f\ x | x \in A\}$, the range of f is $rng\ f = f[\mathcal{U}]$, the preimage of B under f is $f^{-1}[B] = \{x | \ f\ x \in B\}$, the cartesian product is $A \times B = \{(a, b) \mid a \in A, b \in B\}$, and the dependent product is $\Pi x \in A.\ B\ x = \{f | \ \forall x \in A.\ f\ x \in B\ x\}$ and $A \to B = \Pi x \in A. B$. The indicator function $\chi_A\ x = 1$ if $x \in A$ otherwise $\chi_A\ x = 0$. With $f \uparrow x$ we state that f converges monotonically from below to x, this is defined for functions with range $\overline{\mathbb{R}}$ and sets. Hilbert choice is $SOME\ x.\ P\ x$, i.e. $(\exists x.Px) \longrightarrow P(SOME\ x.P\ x)$ holds. For the product spaces we also use the extensional dependent product

$$\Pi_{\!E}\, x \in A.\ B\ x = \{f | \ (\forall x \in A.\ f\ x \in B\ x) \wedge (\forall x \notin A.\ f\ x = \bot)\}$$

We need to enforce having exactly one value outside of the domain, otherwise there is more than one function with the same values on A. We use $t :: \alpha$ to annotate types and $\alpha \Rightarrow \beta$ for function types. These should not be confused with set membership $x \in A$ or the dependent function space $A \to B$, which are predicates and not type annotations. We use $t :: (\alpha :: \text{type\_class})$ to annotate type classes.

The **locale**-command introduces a new locale [4]. We use it to define the concept of algebras, σ-algebras, measure spaces etc.

$$\textbf{locale}\ l = bs + \textbf{fixes}\ x :: \alpha\ \textbf{assumes}\ P_1\ x\ \textbf{and}\ \ldots\ \textbf{and}\ P_n\ x$$

This introduces the locale l with a variable x and the assumptions $P_1, \ldots, P_n$. It inherits the context like variables and assumptions, but also theorems, abbreviations, definitions, setup for the proof methods and more from bs. We get the

[1] http://isabelle.in.tum.de/repos/isabelle/file/tip/src/HOL/Probability

theorems about a specific instantiation $l\ x$ by proving $P_1\ x \wedge \cdots \wedge P_n\ x$. When we prove a theorem in a locale we have access to the theorems of *bs*, i.e. a lemma in *algebra* is immediately available in the *sigma_algebra* locale.

3 Extended Reals

The Lebesgue measure $\boldsymbol{\lambda}$ takes infinite values, as there is no real number we can reasonably assign to $\boldsymbol{\lambda}(\mathbb{R})$. So we need a type containing the real numbers and a distinct value for infinity. We introduce the type $\overline{\mathbb{R}}$ as the reals extended with a positive and a negative infinite element.

Definition 1 (Extended reals $\overline{\mathbb{R}}$)

$$\textbf{\textit{datatype}}\ \overline{\mathbb{R}} = \infty \mid (\mathbb{R})_\infty \mid -\infty \qquad\qquad \mathit{real} :: \overline{\mathbb{R}} \Rightarrow \mathbb{R}$$
$$\mathit{real}\ (r_\infty) = r \qquad \mathit{real}\ \infty = 0 \qquad \mathit{real}\ (-\infty) = 0$$

The conversion function *real* restricts the extended reals to the real numbers and maps ±∞ to 0. For the sake of readability we hide this conversion function.

Definition 2 (Order and arithmetic operations on $\overline{\mathbb{R}}$)

$$\begin{array}{rclll}
r_\infty \leq p_\infty & \longleftrightarrow & r \leq p & x \leq \infty & -\infty \leq x \\
-(r_\infty) & = & (-r)_\infty & -(-\infty) = \infty & \\
r_\infty + p_\infty & = & (r+p)_\infty & \infty + x = \infty & x + \infty = \infty \\
r_\infty \cdot p_\infty & = & (r \cdot p)_\infty & \multicolumn{2}{l}{x \cdot \pm\infty = \pm\infty \cdot x = \begin{cases} 0 & \mathit{if}\ \ x = 0 \\ \operatorname{sgn} x \cdot \pm\infty & \mathit{otherwise} \end{cases}}
\end{array}$$

For measure theory it is suitable to define $\infty \cdot 0 = 0$. Using *min* and *max* as join and meet, we get that $\overline{\mathbb{R}}$ is a complete lattice where *bot* is $-\infty$ and *top* is ∞.

Our next step is to define the topological structure on $\overline{\mathbb{R}}$. This is an extension of the topological structure on real numbers. However we need to take care of what happens when ±∞ is in the set.

Definition 3. $\mathit{open}\ A \longleftrightarrow \mathit{open}\ \{r \mid r_\infty \in A\} \wedge$
$(\infty \in A \longrightarrow \exists x. \forall y > x.\ y_\infty \in A) \wedge (-\infty \in A \longrightarrow \exists x. \forall y < x.\ y_\infty \in A)$

From this definition the continuity of $\cdot_\infty$ follows directly. The definition of limits of sequences in Isabelle/HOL is based on topological spaces. This allows us to reuse these definitions and also some of the proofs such as uniqueness of limits. We also verify that the limits and infinite sums on real numbers are the same as the limits and sums on extended reals:

Lemma 1. $(\lambda n. (f\ n)_\infty) \xrightarrow[n\to\infty]{} r_\infty \qquad \longleftrightarrow \qquad (\lambda n.\ f\ n) \xrightarrow[n\to\infty]{} r$

Corollary 1. *If f is summable, then* $\sum_n\ (f\ n)_\infty = (\sum_n f\ n)_\infty$.

Hurd [7] formalizes similar positive extended reals and also defines a complete lattice on them. Our $\overline{\mathbb{R}}$ includes negative numbers and we not only show that it forms a complete lattice but also that it forms a topological space. The complete lattice is used for monotone convergence and the topological space is used to define a Borel σ-algebra on $\overline{\mathbb{R}}$.

4 Measure Theory

We largely follow Bauer's textbook [1] for our formalization of measure theory. An exception is the definition of the Lebesgue integral which is taken from Schilling [12].

4.1 The σ-Algebra

We use records to represent (σ-)algebras and measure spaces. We define measure spaces as extensions to algebras, hence we can use measure spaces as algebras.

$$\begin{aligned}\textbf{record } \alpha\ \mathit{algebra} = {} & \mathit{space} :: \alpha\ \mathit{set} \\ & \mathit{sets} :: \alpha\ \mathit{set}\ \mathit{set}\end{aligned}$$

To represent the algebra $M = (\Omega, \mathcal{A})$ we write $M = (\!|\mathit{space} = \Omega, \mathit{sets} = \mathcal{A}|\!)$. We use this type to introduce the concept of (σ-)algebras. The set Ω is typically but not necessarily the universe $\mathcal{U}$. For probability theory in particular, it is often $[0,1]$ instead of $\mathbb{R}$. The sets in $\mathcal{A}$ are the measurable sets.

Definition 4 (σ-algebra)

$$\begin{aligned}\textbf{locale } \mathit{algebra}\ = {} & \\ \textbf{fixes } & M :: \alpha\ \mathit{algebra} \\ \textbf{assumes } & \mathit{sets}\ M \subseteq \mathcal{P}(\mathit{space}\ M) \\ \textbf{and } & \emptyset \in \mathit{sets}\ M \\ \textbf{and } & \forall a \in \mathit{sets}\ M.\ \mathit{space}\ M - a \in \mathit{sets}\ M \\ \textbf{and } & \forall a, b \in \mathit{sets}\ M.\ a \cup b \in \mathit{sets}\ M\end{aligned}$$

$$\begin{aligned}\textbf{locale } \mathit{sigma\_algebra}\ = {} & \ \mathit{algebra} + \\ \textbf{assumes } & \forall F :: \mathit{nat} \Rightarrow \alpha\ \mathit{set}.\ \mathit{rng}\ F \subseteq \mathit{sets}\ M \longrightarrow (\textstyle\bigcup_i F\ i) \in \mathit{sets}\ M\end{aligned}$$

The easiest way to define a σ-algebra (other than the power set) is to give a generator $\mathcal{G}$ and use the smallest σ-algebra containing $\mathcal{G}$ (called its *σ-closure*).

Definition 5 (σ-closure). *sigma_sets $\mathcal{G}$ Ω denotes the smallest superset of $\mathcal{G}$ containing $\emptyset$ and is closed under Ω-complement and countable union.*

$$\begin{aligned}&\textbf{inductive } \mathit{sigma\_sets} \textbf{ for } \mathcal{G} \textbf{ and } \Omega \textbf{ where} \\ &\quad a \in \mathcal{G} \longrightarrow a \in \mathit{sigma\_sets}\ \Omega\ \mathcal{G} \\ &\quad \emptyset \in \mathit{sigma\_sets}\ \Omega\ \mathcal{G} \\ &\quad a \in \mathit{sigma\_sets}\ \Omega\ \mathcal{G} \longrightarrow \Omega - a \in \mathit{sigma\_sets}\ \Omega\ \mathcal{G} \\ &\quad \mathit{rng}\ (F :: \mathit{nat} \Rightarrow \alpha\ \mathit{set}) \subseteq \mathit{sigma\_sets}\ \Omega\ \mathcal{G} \longrightarrow (\textstyle\bigcup_i F\ i) \in \mathit{sigma\_sets}\ \Omega\ \mathcal{G}\end{aligned}$$

$$\mathit{sigma}\ M = (\!|\mathit{space} = \mathit{space}\ M, \mathit{sets} = \mathit{sigma\_sets}\ (\mathit{space}\ M)\ (\mathit{sets}\ M)|\!)$$

We define the σ-closure inductively to get a nice induction rule. Then we show that it actually is the smallest σ-algebra containing $\mathcal{G}$.

Lemma 2. *The sigma operator generates a σ-algebra.*
$\mathit{sets}\ M \subseteq \mathcal{P}(\mathit{space}\ M) \longrightarrow \mathit{sigma\_algebra}\ (\mathit{sigma}\ M)$

Lemma 3. *If $\mathcal{G} \subseteq \mathcal{P}(\Omega)$ then*
$\mathit{sigma\_sets}\ \Omega\ \mathcal{G} = \bigcap \left\{ B \supseteq \mathcal{G} \mid \mathit{sigma\_algebra}\ (\!|\mathit{space} = \Omega, \mathit{sets} = B|\!) \right\}$

Measurable Functions. When preimages of measurable sets in M_2 under f are measurable sets in M_1 we say f is M_1-M_2-*measurable.* We use the function-type to represent them, but restrict it to the functions from *space* M_1 to *space* M_2. We also need to intersect the preimage under f with *space* M_1.

Definition 6 (Measurable). *measurable* M_1 M_2 =
$\{f \in \mathit{space}\ M_1 \to \mathit{space}\ M_2 |\ \forall A \in \mathit{sets}\ M_2.\ f^{-1}[A] \cap \mathit{space}\ M_1 \in \mathit{sets}\ M_1\}$

When M_2 is generated by a σ-closure it is enough to show that it is measurable on the generator:

Lemma 4. *If sets* $G \subseteq \mathcal{P}(\mathit{space}\ G)$ *and* $f \in \mathit{measurable}\ M_1\ G$ *then*
$f \in \mathit{measurable}\ M_1\ (\mathit{sigma}\ G)$.

Borel σ-Algebra. The σ-algebra generated by the open sets of a topological space is called a Borel σ-algebra. In Isabelle/HOL topological spaces form a type class defining the open sets. Instances are Euclidean spaces (hence $\mathbb{R}$) and $\overline{\mathbb{R}}$.

Definition 7 (Borel sets)
$\mathit{borel} = \mathit{sigma}\ (\!|\mathit{space} = \mathcal{U} :: (\alpha :: \mathit{topological\_space})\ \mathit{set}, \mathit{sets} = \{S |\ \mathit{open}\ S\}|\!)$

As a first step we show that the Borel sets on real numbers are not only generated by all the open sets, but also by all the intervals $]-\infty, a[$. Then we show the Borel measurability of arithmetic operations, min, max, etc. To show the measurability of these operations on $\overline{\mathbb{R}}$ we first show that $\cdot_\infty$ and *real* are Borel-Borel-measurable (which follows from their continuity). The operations on $\overline{\mathbb{R}}$ are compositions of $\cdot_\infty$ and *real* with operations on real numbers. We use "M-measurable" as abbreviation for "M-Borel-measurable."

Dynkin Systems. We use Dynkin systems to prove the uniqueness of measures. Compared with σ-algebras, they are only closed under countable unions *if the sets are disjoint.*

Definition 8. $\mathit{disjoint}\ F \longleftrightarrow (\forall i, j.\ i \neq j \longrightarrow F\ i \cap F\ j = \emptyset)$

Definition 9 (Dynkin system)

locale *dynkin_system* =
  fixes $D :: \alpha\ \mathit{algebra}$
  assumes $\mathit{sets}\ D \subseteq \mathcal{P}(\mathit{space}\ D)$
    and $\emptyset \in \mathit{sets}\ D$
    and $\forall a \in \mathit{sets}\ D.\ \mathit{space}\ D - a \in \mathit{sets}\ D$
    and $\forall F.\ \mathit{disjoint}\ F \wedge \mathit{rng}\ F \subseteq \mathit{sets}\ D \longrightarrow (\bigcup_i F\ i) \in \mathit{sets}\ D$

Definition 10 (Closed under intersection)
$\cap\text{-}\mathit{stable}\ G \longleftrightarrow (\forall A, B \in \mathit{sets}\ G.\ A \cap B \in \mathit{sets}\ G)$

Dynkin systems are now used to prove Dynkin's lemma, which helps to generalize statements about all sets of a $\cap$-*stable* set to the σ-closure of that set. We use Dynkin's lemma to prove the uniqueness of measures.

Theorem 1 (Dynkin's lemma). *For any Dynkin system* D *and* $\cap$-*stable system* G, *if sets* $G \subseteq \mathit{sets}\ D \subseteq \mathit{sets}\ (\mathit{sigma}\ G)$, *then* $\mathit{sigma}\ G = D$.

4.2 Measure Spaces

A measure space is a σ-algebra extended with a measure which maps measurable sets to nonnegative, possibly infinite *measure* values. We introduce a new type *measure_space* which extends the *algebra* record. We represent measure values with $\overline{\mathbb{R}}$, and abbreviate $\Omega = \mathit{space}\ M$, $\mathcal{A} = \mathit{sets}\ M$, and $\mu = \mathit{measure}\ M$.

Definition 11 (Measure space)

$$\begin{array}{l} \textbf{record}\ \mathit{measure\_space} = \alpha\ \mathit{algebra}\ +\ \mathit{measure} :: (\alpha\ \mathit{set}) \Rightarrow \overline{\mathbb{R}} \\ \textbf{locale}\ \mathit{measure\_space} = \mathit{sigma\_algebra}\ M\ \textbf{for}\ M :: \alpha\ \mathit{measure\_space}\ + \\ \quad \textbf{assumes}\ \mu\ \emptyset = 0 \\ \qquad \textbf{and}\ \forall A \in \mathcal{A}.\ 0 \leq \mu\ A \\ \qquad \textbf{and}\ \forall F.\ \mathit{disjoint}\ F \wedge \mathit{rng}\ F \subseteq \mathcal{A} \longrightarrow \mu\ \left(\bigcup_i F\ i\right) = \sum_i \mu\ (F\ i) \end{array}$$

In the remaining sections we fix the measure space M. We prove the additivity, monotonicity, and continuity from above and below for measures. For proving the existence of a measure we provide Caratheodory's theorem, which was ported from Hurd [6] and Coble [2].

Theorem 2 (Caratheodory). *Assume G is an algebra, and let be f a function such that f is nonnegative on $\mathcal{A}$, $f\ \emptyset = 0$, and is f countably additive, i.e.,*

$$\forall F.\ \mathit{disjoint}\ F \wedge \mathit{rng}\ F \subseteq \mathit{sets}\ G \longrightarrow f\ \left(\bigcup_i F\ i\right) = \sum_i f\ (F\ i)$$

then there exists a ν s.t. $\forall A \in \mathit{sets}\ G.\ \nu\ A = f\ A$ and $\mathit{sigma}\ G(\!|\mathit{measure} := \nu|\!)$ is a measure space.

For our purposes to formalize product measures and to equate the products of the Lebesgue measure, we prove the uniqueness of measures.

Theorem 3 (Uniqueness of measures). *Assume*

- *μ and ν are two measures on sigma G*
- *G is $\cap$-stable*
- *C is a σ-finite cover of G: $\mathit{rng}\ C \subseteq \mathit{sets}\ G$, $C \uparrow \mathit{space}\ G$, and $\forall i.\ \mu(C\ i) < \infty$*
- *μ and ν are equal on G: $\forall X \in \mathit{sets}\ G.\ \mu\ X = \nu\ X$*

then μ and ν are equal on sigma G.

An important class of measure spaces are σ-finite measure spaces. It requires a sequence of finitely measurable sets which cover the entire space. The product measure and the Radon-Nikodým theorem assume a σ-finite measure.

Definition 12 (σ-finite measure space)

$$\begin{array}{l} \textbf{locale}\ \mathit{sigma\_finite\_measure} = \mathit{measure\_space}\ + \\ \quad \textbf{assumes}\ \exists F.\ \mathit{rng}\ F \subseteq \mathcal{A} \wedge F \uparrow \Omega\ \wedge \forall i.\ \mu\ (F\ i) < \infty \end{array}$$

Almost Everywhere. Often predicates on measure spaces do not hold for all elements in a measure space, but the elements where they do not hold form a subset of a null set. In textbooks this is often written without an explicitly quantified variable but rather with an appended "a.e." (standing for "almost everywhere"). We use a syntax with an explicit binder.

Definition 13 (Almost everywhere)
$(\mathit{AE}\, x.\ P\ x) \longleftrightarrow \exists N \in \mathcal{A}.\ \{x \in \Omega |\ \neg\ P\ x\} \subseteq N \wedge \mu\ N = 0$

The definition of almost everywhere in [6] and [10] assumes that $\{x \in \Omega |\ \neg\ P\ x\}$ is a null set, i.e. it is also measurable.

Theorem 4 (AE modus ponens)
$(\mathit{AE}\, x.\ P\ x) \longrightarrow (\mathit{AE}\, x.\ P\ x \longrightarrow Q\ x) \longrightarrow (\mathit{AE}\, x.\ Q\ x)$

Our relaxed definition requires no measurability of Q in the *modus ponens* rule of almost everywhere.

Theorem 5. $(\forall x \in \Omega.\ P\ x) \longrightarrow (\mathit{AE}\, x.\ P\ x)$

Let us take a look at the statement $(\mathit{AE}\, x.\ f\ x < g\ x) \longrightarrow (\mathit{AE}\, x.\ f\ x \leq g\ x)$. This can be directly solved by AE modus ponens and theorem 5. The measurability of f and g is not required.

4.3 Lebesgue Integral

The definition of the Lebesgue integral requires the concept of simple functions. A simple function is a Borel-measurable step function (i.e. its range is a finite set), or equivalently a step function where the preimage of every singleton set containing an element of the range is measurable. The second formulation has the advantage that the definition does not require the notion of Borel σ-algebras and is thus more general, as it allows arbitrary ranges. The predicate *simple_function* is defined as follows:

Definition 14 (Simple function)

$$\mathit{simple\_function}\ f \longleftrightarrow \mathit{finite}\ f[\Omega] \wedge \forall x \in f[\Omega].\ f^{-1}[\{x\}] \cap \Omega \in \mathcal{A}$$

While we use this definition only for functions $f\ ::\ \alpha \Rightarrow \overline{\mathbb{R}}$, this is a nice characterisation for finite random variables in probability theory. When the range of f is $\overline{\mathbb{R}}$ it is also representable as sum:

Lemma 5

$$\forall x \in \Omega.\ f\ x = \sum_{y \in f[\Omega]} y \cdot \chi_{f^{-1}[\{y\}] \cap \Omega}\ x$$

This already suggests the definition of the integral $\int^S$ of a simple function f with respect to the measure space M:

Definition 15 (Simple integral). *Let f be a simple function.*

$$\int^S f\ dM = \sum_{y \in f[\Omega]} y \cdot \mu(f^{-1}[\{y\}] \cap \Omega)$$

To state the definition of the integral of functions $f :: \alpha \Rightarrow \overline{\mathbb{R}}$, simple functions have to be used as approximations from below of f. Then the integral is defined as the supremum of all the simple integrals of the approximations.

Definition 16 (Positive integral)

$$\int^+ f\ dM = \sup\left\{\int^S g\ dM \;\middle|\; g \le f^+ \wedge \mathit{simple\_function}\ g\right\}$$

The function f^+ is the nonnegative part of f, i.e. f^+ is zero when f is negative, otherwise it is equal to f. Hence the positive integral is equal when the integrating functions are almost everywhere equal. Finally integration can be defined for functions $f :: \alpha \Rightarrow \mathbb{R}$ as usual.

Definition 17 (Lebesgue Integrability and Integral)

$$\begin{aligned}
\mathit{integrable}\ M\ f \longleftrightarrow{}& f \in \mathit{measurable}\ M\ \mathit{borel} \wedge \\
& \left(\int^+ x.\ (f\ x)_\infty\ dM\right) < \infty \wedge \left(\int^+ x.\ (-f\ x)_\infty\ dM\right) < \infty \\
\int f\ dM \ ={}& \left(\int^+ x.\ (f\ x)_\infty\ dM\right) - \left(\int^+ x.\ (-f\ x)_\infty\ dM\right)
\end{aligned}$$

(Note that explicit type conversions from $\overline{\mathbb{R}}$ to $\mathbb{R}$ have been omitted in this definition for the sake of readability.)

Remark: Textbooks usually write $\int f d\mu(x)$, where we instead specify the entire measure space M and optionally bind the variable x directly after the integral symbol, $\int x.\, f\, x\, dM$. If no variable is needed we write $\int f\, dM$, and a restricted integral is abbreviated as $\int x \in A. f\ x\ dM = \int x. f\ x \cdot \chi_A\ x\ dM$.

Many proofs of properties about the integral follow the scheme of the definitions and first establish the desired property for $\int^S$, then for $\int^+$, and eventually for $\int$. The monotonicity of the integral is proven this way, for example.

Lemma 6 (Monotonicity). *If f and g are measurable functions, then*

$$(\mathit{AE}\, x.\ f\ x \le g\ x) \longrightarrow \int f\, dM \le \int g\, dM$$

Another way of constructing proofs about Borel-measurable functions $u :: \alpha \Rightarrow \overline{\mathbb{R}}$ is: first, prove the desired property about finite simple functions, then, prove that the property is preserved under the pointwise monotone limit of functions. For this to work, we need a lemma stating that every Borel-measurable function $u :: \alpha \Rightarrow \overline{\mathbb{R}}$ can be seen as the limit of a monotone sequence of finite simple functions.

Lemma 7. *Let u be a nonnegative and measurable function.*

$$\exists f.\ (\forall i.\ \mathit{simple\_function}\ (f\ i) \wedge (\forall x \in \Omega.\ 0 \le\ f\ i\ x\ \ne \infty)) \wedge f \uparrow u$$

To use this with the Lebesgue integral, there is a compatibility theorem, called the monotone convergence theorem, which allows switching the supremum operator and the positive integral.

Lemma 8 (Monotone convergence theorem). *Let* $f :: \mathbb{N} \Rightarrow \alpha \Rightarrow \overline{\mathbb{R}}$ *be a sequence of nonnegative Borel-measurable functions, such that* $\forall i. \forall x \in \Omega.\ f\ i\ x \leq f\ (i+1)\ x$*. Then it holds that:*

$$\sup\ i.\ \int^{+} f\ i\ dM = \int^{+} (\sup\ i.\ f\ i)\ dM$$

The Monotone convergence theorem is used in the proof of Fubini's theorem. Another useful convergence theorem is the dominated convergence theorem. It can be used when the monotonicity of the function sequence does not hold.

Lemma 9 (Dominated convergence theorem). *Let* $u :: \mathbb{N} \Rightarrow \alpha \Rightarrow \mathbb{R}$ *be a sequence of integrable functions,* $w :: \alpha \Rightarrow \mathbb{R}$ *be an integrable function, and* $v :: \alpha \Rightarrow \mathbb{R}$ *be a function. If* $(\forall i.\ |u\ i\ x| \leq w\ x)$ *and* $(\lambda i.\ u\ i\ x) \longrightarrow_{\infty} v\ x$ *for all* $x \in \Omega$ *then integrable* $M\ v$ *and* $\big(\lambda i.\ \int u\ i\ dM\big) \longrightarrow_{\infty} \int v\ dM$*.*

To transfer results about integrals from one measure space to another one, the following transformation lemma can be used.

Lemma 10. *If* T *is* M*-*M'*-measurable and measure* $M'\ A$ *equals* $\mu\ (T^{-1}[A] \cap \Omega)$ *for all* $A \in$ *sets* M' *and* f *is* M'*-integrable, then* $f \circ T$ *is* M'*-integrable and*

$$\int f\ dM' = \int f \circ T\ dM$$

4.4 Radon-Nikodým Derivative

The Radon-Nikodým theorem states that for each measure ν that is absolutely continuous on M there exists an a.e.-unique density function to represent ν on M. This is needed to define *conditional expectation* in probability theory and *mutual information* in information theory. In this section we assume that M is σ-finite.

Definition 18 (Radon-Nikodým derivative)

$$\begin{aligned}&\mathit{RN\_deriv}\ M\ \nu = \mathit{SOME}\ f \in \mathit{measurable}\ M\ \mathit{borel}.\\ &\quad (\forall x \in \Omega.\ 0 \leq f\ x) \wedge \Big(\forall X \in \mathcal{A}.\ \nu\ X = \Big(\int^{+} x \in X.\ f\ x\ dM\Big)\Big)\end{aligned}$$

To work with this definition we need to prove the existence of such a function.

Theorem 6 (Radon-Nikodým). *If* ν *is a measure on* M *and* ν *is absolutely continuous w.r.t.* M*, i.e.,* $\forall A \in \mathcal{A}.\ \mu\ A = 0 \longrightarrow \nu\ A = 0$ *then*

$$\mathit{RN\_deriv}\ M\ \nu \in \mathit{measurable}\ M\ \mathit{borel}$$

$$\wedge\ \forall X \in \mathcal{A}.\ \nu\ X = \Big(\int^{+} x \in X.\ \mathit{RN\_deriv}\ M\ \nu\ x\ dM\Big)$$

The next theorem shows that two functions are a.e.-equal when they are equal on all measurable sets, hence follows the uniqueness of *RN_deriv*.

Theorem 7. *If* f *and* g *are nonnegative and* M*-measurable and* $\forall A \in \mathcal{A}.\ \Big(\int^{+} x \in A.\ f\ x\ dM\Big) = \Big(\int^{+} x \in A.\ g\ x\ dM\Big)$ *then* $(\mathit{AE}\ x.\ f\ x = g\ x)$

4.5 Product Measure and Fubini's Theorem

We first introduce the binary product of measure spaces, and later extend this to arbitrary, finite products of measure spaces.

Binary Product Measure. The definition of a measure on a binary product σ-algebra is straightforward. All we need to do is compose the integrals of both measure spaces. With Fubini's theorems we later show that the result is independent of the order of integration.

Definition 19

$$\begin{aligned}
\mathit{bin\_algebra}_G &:: \alpha\ \mathit{measure\_space} \Rightarrow \beta\ \mathit{measure\_space} \\
&\qquad \Rightarrow (\alpha \times \beta)\ \mathit{measure\_space} \\
\mathit{bin\_algebra}_G\ M_1\ M_2 &= (\!|\mathit{space} = \mathit{space}\ M_1 \times \mathit{space}\ M_2, \\
&\qquad \mathit{sets} = \{A \times B | A \in \mathit{sets}\ M_1, B \in \mathit{sets}\ M_2\}, \\
&\qquad \mathit{measure} = \int^{+} x.\ \left(\int^{+} y.\ \chi_{A \times B}\ (x,y)\ dM_2\right) dM_1|\!) \\
M_1 \bigotimes\nolimits_m M_2 &= \mathit{sigma}(\mathit{bin\_algebra}_G\ M_1\ M_2)
\end{aligned}$$

In this section we assume that M_1 and M_2 are σ-finite measure spaces. We verify the definition of the binary product measure by applying the measure to an element $A \times B$ from the generating set of $M_1 \bigotimes_m M_2$.

Lemma 11. *If $A \in \mathit{sets}\ M_1$ and $B \in \mathit{sets}\ M_2$ then* $\mathit{measure}\ (M_1 \bigotimes_m M_2)\ (A \times B) = \mathit{measure}\ M_1\ A \cdot \mathit{measure}\ M_2\ B$.

Lemma 12. *Show the measurability of the cut $\{y|(x,y) \in Q\}$ for all $Q \in \mathit{sets}\ (M_1 \bigotimes_m M_2)$ and all x.*

$$\{y|(x,y) \in Q\} \in \mathit{sets}\ M_2 \tag{1}$$

$$(\lambda x.\ \mathit{measure}\ M_2\ \{y|(x,y) \in Q\}) \in \mathit{measurable}\ M_1\ \mathit{borel} \tag{2}$$

$$\mathit{measure}\ (M_1 \bigotimes\nolimits_m M_2)\ Q = \int^{+} x.\ \mathit{measure}\ M_2\ \{y|(x,y) \in Q\}\ dM_1 \tag{3}$$

Theorem 8. *$\mathit{sigma\_finite\_measure}\ (M_1 \bigotimes_m M_2)$*

Fubini's Theorem. From the product measure we get directly to the fact that integrals on σ-finite measure spaces are commutative.

Lemma 13. *If f is $M_1 \bigotimes_m M_2$-measurable then*

$$\left(\lambda x. \int^{+} y.\ f\ (x,y)\ dM_2\right) \in \mathit{measurable}\ M_1\ \mathit{borel} \quad \mathit{and}$$

$$\int^{+} x.\ \left(\int^{+} y.\ f\ (x,y)\ dM_1\right) dM_2 = \int^{+}\ f\ d\,(M_1 \bigotimes\nolimits_m M_2)\ .$$

With theorem 3 we show that the pair swap function $(\lambda(x,y).(y,x))$ is measure preserving between $M_1 \bigotimes_m M_2$ and $M_2 \bigotimes_m M_1$. This allows us to get symmetric variants of (1), (2), and (3) without reproducing a symmetric proof.

Corollary 2 (Fubini's theorem on $\overline{\mathbb{R}}$). *If f is $M_1 \bigotimes_m M_2$-measurable then*

$$\textstyle\int^+ x.\ \left(\int^+ y.\ f\ (x,y)\ dM_2\right)\ dM_1 = \int^+ y.\ \left(\int^+ x.\ f\ (x,y)\ dM_1\right)\ dM_2$$

Lemma 13 can be extended to integrability on real numbers.

Lemma 14. *If f is $M_1 \bigotimes_m M_2$-integrable then*

$$M_1\text{-}AE\, x.\ integrable\ M_2\ (\lambda y.\ f\ (x,y)) \quad and$$

$$\textstyle\int x.\ \left(\int y.\ f(x,y)\ dM_2\right)\ dM_1 = \int f\ d(M_1 \bigotimes_m M_2)\ .$$

Finally, we prove Fubini's theorem by this lemma and its symmetric variant.

Corollary 3 (Fubini's theorem). *If f is $M_1 \bigotimes_m M_2$-integrable then*

$$\textstyle\int x. \left(\int y.\ f\ (x,y)\ dM_2\right)\ dM_1 = \int y. \left(\int x.\ f\ (x,y)\ dM_1\right)\ dM_2\ .$$

Product Measures. Product spaces are modeled as function space, i.e. the space of dependent products. In this section we assume $M\ i$ is a σ-finite measure space for all i. Product spaces can also be defined on arbitrary index sets I, however this holds only on probability spaces. We assume a finite index set I.

Definition 20

$$\begin{array}{l}
prod\_algebra_G \qquad :: \iota\ set \Rightarrow (\iota \Rightarrow \alpha\ algebra) \Rightarrow (\iota \Rightarrow \alpha)\ algebra \\
prod\_algebra_G\ I\ M = (\!|space = (\Pi_E i \in I.\ space\ (M\ i)), \\
\qquad\qquad\qquad sets = \left\{ (\Pi_E i \in I.\ E\ i) \middle| E.\ \forall i \in I.\ E\ i \in sets\ (M\ i) \right\} |\!) \\
\Pi_m\ i \in I.\ M\ i \qquad = sigma\ (prod\_algebra_G\ I\ M)(\!|measure := SOME\ \nu. \\
\quad sigma\_finite\_measure\ (sigma\ (prod\_algebra_G\ I\ M)(\!|measure := \nu|\!)) \wedge \\
\quad \forall E.\ (\forall i \in I.\ E\ i \in sets\ (M\ i)) \\
\qquad \longrightarrow \nu\ (\Pi_E i \in I.\ E\ i) = \prod_{i \in I} measure\ (M\ i)\ (E\ i)|\!)
\end{array}$$

We abbreviate $P_I = (\Pi_m\ i \in I.\ M\ i)$ and $\pi_I = measure\ P_I$. The definition of P_I takes *sigma* $(prod\_algebra_G\ I\ M)$ and extends it with *some* measure ν which forms a σ-finite measure space and which is uniquely defined on $prod\_algebra_G\ I\ M$, i.e., the generating set. These properties only holds for P_I when such a measure function exists, we prove the existence by induction over the finite set I.

Theorem 9. *If I is a finite set then sigma_finite_measure P_I and*
$\forall E.\ (\forall i.\ E\ i \in sets\ (M\ i)) \longrightarrow \pi_I\ (\Pi_E i \in I.\ E\ i) = \prod_{i \in I} measure\ (M\ i)\ (E\ i)$

We use $merge\ I\ J = (\lambda(x,y)\ i.\ if\, i \in I\ then\, x\ i\ else\ if\, i \in J\ then\, y\ i\ else\, \bot)$ as measure preserving function from $P_I \bigotimes_m P_J$ to $P_{I \cup J}$.

Lemma 15. *If I and J are two disjoint finite sets and $A \in sets\ P_{I \cup J}$ then*

$$\pi_{I \cup J}\ A = measure\ (P_I \bigotimes\nolimits_m P_J)\ ((merge\ I\ J)^{-1}[A] \cup space\ (P_I \bigotimes\nolimits_m P_J))$$

A finite index set I' is either represented as $I' = I \cup J$, wih I and J finite, or $I' = \{i\}$. We give rules how to handle integrals in both cases, this allows us to iterate the Lebesgue integral on nonnegative functions in an inductive proof.

Lemma 16. *If I and J are disjoint finite sets and f is $P_{I\cup J}$-measurable then*

$$\int^+ f\ dP_{I\cup J} = \int^+ x.\ \left(\int^+ y.\ f\ (\mathit{merge}\ I\ J\ (x, y))\ dP_J\right)\ dP_I$$

Lemma 17. *If f is $M\ i$-measurable then*

$$\int^+ x.\ f\ (x\ i)\ dP_{\{i\}} = \int^+ f\ d(M\ i)$$

We extend these two lemmas to Lebesgue integrable functions. This helps us to prove the distributivity of multiplication and integration by induction on I.

Corollary 4. *If $I \neq \emptyset$ is finite and $f\ i$ is $M\ i$-integrable for all $i \in I$ then*

$$\int x.\ \left(\prod_{i\in I} f\ i\ (x\ i)\right) dP_I = \prod_{i\in I} \left(\int (f\ i)\ d(M\ i)\right)\ .$$

4.6 Lebesgue Measure

We have now formalized the concepts of measure spaces, Lebesgue integration and product spaces. An important measure space is the one on $\mathbb{R}$, where each interval $[a, b]$ has as measure value the length of the interval, $b - a$. The Borel σ-algebra is generated by these intervals. The corresponding measure is called the Lebesgue-Borel measure, its completion is the Lebesgue measure.

Instead of following the usual construction of the Lebesgue measure as the σ-extension of an interval measure we use the gauge integral[2] available in the multivariate analysis in Isabelle/HOL.[3] The gauge integral is an extension of the Riemann and also of the Lebesgue integral on Euclidean vector spaces. In Isabelle/HOL the predicate *integrable_on* $A\ f$ states that the function f is gauge integrable on the set A, in which case the gauge integral of f on the set A has the real value *integral* $A\ f$. The gauge measure of a set A is the gauge integral of the constant 1 function on A.

Since the gauge measure is restricted to finitely measurable sets, it cannot be used directly as Lebesgue measure. However we can measure the indicator function χ_A on the intervals $[-n, n]$ for all natural numbers n. When χ_A is measurable on all intervals, we define it as Lebesgue measurable and the Lebesgue measure is the supremum of the gauge measures for all intervals $[-n, n]$. To define the Lebesgue measure on multidimensional Euclidean spaces we use hypercubes $\{x | \forall i.\ |x_i| \leq n\}$. The σ-algebra of the Lebesgue measure on a Euclidean space α consists of all $A{::}\alpha\ \mathit{set}$ which are gauge measurable on all intervals.

Definition 21 (Lebesgue measure)

$$\begin{aligned} \mathit{lebesgue}_\alpha = (\!|\ & \mathit{space} = \mathcal{U}, \\ & \mathit{sets} = \{A | \forall n.\ \mathit{integrable\_on}\ \{x | \forall\ i.\ |x_i| \leq n\}\ (\chi_A)\} \\ & \mathit{measure} = \sup\ n.\ \mathit{integral}\ \{x | \forall\ i.\ |x_i| \leq n\}\ (\chi_A) |\!) \end{aligned}$$

[2] The gauge integral is also called the Henstock-Kurzweil integral.

[3] The multivariate analysis in Isabelle/HOL is ported from a later version of [5].

Theorem 10. *The Lebesgue measure forms a σ-finite measure space.*
sigma_finite_measure $\textit{lebesgue}_\alpha$

From the definition of the Lebesgue measure it is easy to see that all Lebesgue measurable simple functions whose integral is finite are also gauge integrable. With the monotone convergence of the gauge integral we show that all nonnegative Lebesgue measurable functions with a finite integral are gauge integrable. And finally we show that all Lebesgue integrable functions are gauge integrable.

Theorem 11. *If f is Lebesgue integrable then integrable_on* $\mathcal{U}\ f$ *and* $\textit{integral}\,\mathcal{U} f = \int f\ d(\textit{lebesgue}_\alpha)$.

We know that lebesgue_α is a σ-algebra, and since all intervals $[a, b]$ are Lebesgue measurable all Borel sets are Lebesgue measurable.

Lemma 18. $A \in \textit{sets borel} \longrightarrow A \in \textit{sets lebesgue}_\alpha$

We introduce the Lebesgue-Borel measure by changing the measurable sets from the Lebesgue sets to the Borel sets.

Definition 22 (Lebesgue-Borel measure). $\boldsymbol{\lambda}_\alpha = \textit{lebesgue}_\alpha (\!| \textit{sets} := \textit{sets borel} |\!)$

Theorem 12. *sigma_finite_measure* $\boldsymbol{\lambda}_\alpha$

With theorem 3 we know that $\boldsymbol{\lambda}_\alpha$ is equal to other measures introduced on the Borel sets and based on the interval length. The Lebesgue-Borel measure is defined as a sub-σ-algebra of the Lebesgue measure, hence Lebesgue-Borel integrability induces gauge integrability.

Theorem 13. *If f is $\boldsymbol{\lambda}_\alpha$-integrable then integrable_on* $\mathcal{U} f$ *and* $\textit{integral}\,\mathcal{U} f = \int f d\boldsymbol{\lambda}_\alpha$.

Euclidean Vector Spaces and Product Measures. We relate the Euclidean space α with the n-dimensional Lebesgue measure $\boldsymbol{\lambda}^n = (\Pi_m\ i \in \{1, \ldots, n\}.\ \boldsymbol{\lambda}_\mathbb{R})$. The function $\textit{p2e} :: (\mathbb{N} \Rightarrow \mathbb{R}) \Rightarrow \alpha$ maps functions to vectors with $(\textit{p2e}\ f)_i = f\ i$. Theorem 3 helps us to show that it is measure preserving between $\boldsymbol{\lambda}^{\mathcal{D}(\alpha)}$ and α.[4]

Lemma 19. *Any $\boldsymbol{\lambda}_\alpha$-measurable set A satisfies*

$$\textit{measure}\ \boldsymbol{\lambda}_\alpha\ A = \textit{measure}\ \boldsymbol{\lambda}^{\mathcal{D}(\alpha)}\ (\textit{p2e}^{-1}[A] \cap \textit{space}\ \boldsymbol{\lambda}^{\mathcal{D}(\alpha)}).$$

From this follows the equivalence of integrals.

Theorem 14. *If f is $\boldsymbol{\lambda}_\alpha$-measurable then*

$$\begin{array}{rcl} \int^+ f\ d\boldsymbol{\lambda}_\alpha & = & \int^+ x.\ f\ (\textit{p2e}\ x)\ d\boldsymbol{\lambda}^{\mathcal{D}(\alpha)} \\ \textit{integrable}\ \boldsymbol{\lambda}_\alpha\ f & \longleftrightarrow & \textit{integrable}\ \boldsymbol{\lambda}^{\mathcal{D}(\alpha)}\ (f \circ \textit{p2e}) \\ \int f\ d\boldsymbol{\lambda}_\alpha & = & \int x.\ f\ (\textit{p2e}\ x)\ d\boldsymbol{\lambda}^{\mathcal{D}(\alpha)} \end{array}$$

The Euclidean vector space formalizations in Isabelle/HOL include the dimensionality in the vector type. Here it is not possible to use induction over the dimensionality of the Euclidean space. With theorems 13 and 14 we equate the gauge integral to the Lebesgue integral over $\boldsymbol{\lambda}^n$, we then use induction over n.

[4] $\mathcal{D}(\alpha)$ is the dimension of the euclidean space α.

Table 1. Overview of the current formalizations of measure theory

	Hurd	Richter	Coble	Mhamdi	Lester	PVS	Mizar	HOL-Light	Isabelle
$\overline{\mathbb{R}}$					✓	✓	✓		✓
Borel (open)				✓	✓	✓			✓
Integral		✓	✓	✓	✓	✓	✓	✓	✓
$\boldsymbol{\lambda}$	$[0, 1]$					✓	✓	✓	✓
Products						✓		$\mathbb{R}^{\mathcal{D}(\alpha+\beta)}$	✓
Dynkin							✓		✓

5 Discussion

Most measure theory textbooks assume that product spaces are built by iterating binary products and that the Euclidean space is equivalent to the product of the Lebesgue measure. In our case these are three different types, which we need to relate. Using theorem 3 we show the equivalence of measure spaces of theses types. This not only helps us to transfer between different types but also to avoid repeated proofs. For example Fubini's theorem needs the symmetric variant of some theorems. Instead of repeating these proofs as the text books suggest we show that the measure is equal under the pair swap function.

We also diverge from text books by directly constructing the binary product measure and the Lebesgue measure. Usually text books show the existence of a measure and then choose one meeting the specification. This is difficult in theorem provers as the definition is not usable until the existence is proven. Otherwise, we prefer to stay close to the standard formalizations of measure theory concepts. Sometimes this requires more work if we only want to prove one specific lemma, but it is easier to find textbook proofs usable for formalization.

Locales as mechanism for theory interpretation are convenient when proving the Radon-Nikodým theorem and product measures. We instantiate measure spaces restricted to sets obtained in the proof. By interpretation inside the proof we have full access to the automation and lemmas provided by the locale.

Type classes simplify the introduction of $\overline{\mathbb{R}}$ as it allows us to reuse syntax and some theorems about lattices, arithmetic operations, topological spaces, limits, and infinite series. We use the topological space type class to define the Borel σ-algebra. This allows us to state theorems about Borel sets for $\mathbb{R}$ and $\overline{\mathbb{R}}$.

6 Related Work

Our work started as an Isabelle/HOL port of the HOL4 formalization done by Coble [2]. We later reworked most of it to use the extended reals as measure values and open sets as generator for the Borel σ-algebra. We also changed the definition of the Lebesgue integral to the one found in Schilling's textbook [12]. We define the integral of f as the supremum of all simple functions bounded by f. Coble used the limit of the simple functions converging to f.

Table 1 gives an overview of the current formalizations of measure theory we are aware of. The columns list first the work of Hurd [6], Richter [11], Coble [2], Mhamdi et al. [10], and Lester [8]. The second part of the columns list theorem provers or libraries formalizing measure theory. Beginning with the PVS-NASA library,[5] the Mizar Mathematical Library (MML), the multivariate analysis found in HOL Light and finally the work presented in this paper. Mhamdi et al. represents the current state of HOL4, hence HOL4 is not listed. The rows correspond to different measure theoretic concepts and features.

Hurd [6] formalizes a measure space on infinite boolean streams isomorphic to the Lebesgue measure on the unit interval $[0, 1]$. His positive extended reals [7] are unrelated to this. Richter [11] formalizes the Lebesgue integral in Isabelle/HOL and uses it together with Hurd's bitstream measure. Richter introduces the Borel σ-algebra, but only on right-bounded intervals in $\mathbb{R}$.

Coble [2] uses product spaces and the Radon-Nikodým derivative on finite sets to define mutual information for his formalization of information theory. He ports Richter's formalization of the Lebesgue integral to HOL4 and generalizes the definition of σ-algebras to be defined on arbitrary spaces $\Omega \neq \mathcal{U}$. While his formalizations of the Lebesgue integral is on arbitrary measure spaces, he states the Radon-Nikodým theorem and the product measure for finite sets only.

The work by Mhamdi et. al [10] extends Coble's [2]. Their definitions of the Lebesgue integral and Borel σ-algebra are comparable to the ones in this paper. However, they do not formalize measure values as extended real numbers, but only as plain reals. They define a more restricted version of the almost everywhere predicate, and do not give rules for the interaction with logical connectives. They prove Markov's inequality and the weak law of large numbers.

There is also the PVS formalization of topology by Lester [8]. He gives a short overview of the formalized measure theory, which includes measures using extended real numbers, a definition of almost everywhere, Borel σ-algebras on topological spaces, and the Lebesgue integral. In recent developments the PVS-NASA library contains binary product spaces and the proof that the Lebesgue integral extends the Riemann integral. In PVS, abstract reasoning is performed using parametrized theories, similar to our usage of locales.

Endou et. al. [3] proves monotone convergence of the Lebesgue integral in the MML. It also contains measure spaces with extended real numbers, and the Lebesgue measure. Merkl [9] formalized Dynkin systems and Dynkin's lemma in MML, however without a concrete application.

In HOL Light an extended version of Harrison's work [5] introduces gauge integration on finitely-dimensional Euclidean spaces which is similar to the product space of Lebesgue measures. This is then used to define a subset of the Lebesgue measure, missing infinite measure values. The definition of Euclidean spaces $\mathbb{R}^{\mathcal{D}(\alpha)}$ and $\mathbb{R}^{\mathcal{D}(\alpha)}$ allows to create the product $\mathbb{R}^{\mathcal{D}(\alpha+\beta)}$. His theories are now available in Isabelle/HOL and we use them to introduce the Lebesgue measure and to show that Lebesgue integrability implies gauge integrability and that in this case both integrals are equal.

[5] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`

7 Conclusion

The formalizations presented in this paper form the foundations of measure theory. Looking at the table of contents of Bauer's textbook [1] we formalized almost all of the first three chapters. What is missing are the function spaces $\mathcal{L}^p$, stochastic convergence, and the convolution of finite Borel measures. Isabelle supported us with its type classes allowing to reuse definitions and theorems for limits and arithmetic operations on $\overline{\mathbb{R}}$. We used Isabelle's locales to introduce the concepts of the different set systems and spaces used in measure theory.

With product spaces and the Radon-Nikodým derivative it is possible to combine the concepts introduced by Hurd [6] and Coble [2]. We can now verify information theoretic properties of probabilistic programs.

This paper is the first to derive the Radon-Nikodým theorem and the multidimensional version of Fubini's theorem. Our next step concerns the development of probability theory. We already formalized conditional expectation, Kullback-Leibler divergence, mutual information, and infinite products measure using the measure theory presented in this paper. The details are available at the URL given on page 2. The future goals concern the formalization of infinite sequences of independent random variables and the central limit theorem as well as Markov chains and Markov decision processes.

Acknowledgments. We would like to thank Lawrence C. Paulson for porting Coble's measure space formalization from HOL4 to Isabelle/HOL and to Robert Himmelmann for formalizing the first iteration of the Lebesgue measure.

References

1. Bauer, H.: Measure and Integration theory. de Gruyter (2001)
2. Coble, A.R.: Anonymity, Information, and Machine-Assisted Proof. Ph.D. thesis, King's College, University of Cambridge (2009)
3. Endou, N., Narita, K., Shidama, Y.: The Lebesgue monotone convergence theorem. Formal Mathematics 16(2), 167–175 (2008)
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.V.: A HOL theory of euclidean space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
6. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis. University of Cambridge (2002)
7. Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in HOL. Theoretical Computer Science 346(1), 96–112 (2005)
8. Lester, D.R.: Topology in PVS: continuous mathematics with applications. In: Proceedings of AFM 2007, pp. 11–20 (2007)
9. Merkl, F.: Dynkin's lemma in measure theory. In: FM, vol. 9(3), pp. 591–595 (2001)

10. 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)
11. Richter, S.: Formalizing integration theory with an application to probabilistic algorithms. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 271–286. Springer, Heidelberg (2004)
12. Schilling, R.L.: Measures, Integrals and Martingales. Cambridge Univ. Press, Cambridge (2005)

Termination of Isabelle Functions via Termination of Rewriting*

Alexander Krauss[1], Christian Sternagel[2], René Thiemann[2],
Carsten Fuhs[3], and Jürgen Giesl[3]

[1] Institut für Informatik, Technische Universität München, Germany
[2] Institute of Computer Science, University of Innsbruck, Austria
[3] LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. We show how to automate termination proofs for recursive functions in (a first-order subset of) Isabelle/HOL by encoding them as term rewrite systems and invoking an external termination prover. Our link to the external prover includes full proof reconstruction, where all necessary properties are derived inside Isabelle/HOL without oracles. Apart from the certification of the imported proof, the main challenge is the formal reduction of the proof obligation produced by Isabelle/HOL to the termination of the corresponding term rewrite system. We automate this reduction via suitable tactics which we added to the IsaFoR library.

1 Introduction

In a proof assistant based on higher-order logic (HOL), such as Isabelle/HOL [15], recursive function definitions typically require a termination proof. To release the user from finding suitable termination arguments manually, it is desirable to automate these termination proofs as much as possible.

There have already been successful approaches to port and adapt existing termination techniques from term rewriting and other areas to Isabelle [5,12]. They indeed increase the degree of automation for termination proofs of HOL functions. However, these approaches do not cover all powerful techniques that have been developed in term rewriting, e.g., [7,20]. These techniques are implemented in a number of termination tools (e.g., AProVE [9], $\mathsf{T_TT_2}$ [11] and many others) that can show termination of (first-order) term rewrite systems (TRSs) automatically. (In the remainder we use 'termination tool' exclusively to refer to such fully automatic and external provers.) Instead of porting further proof techniques to Isabelle, we prefer to use the existing termination tools, giving direct access to an abundance of methods and their efficient implementations.

Using termination tools inside proof assistants has been an open problem for some time and is often mentioned as future work when discussing certification of termination proofs [3,6]. However, this requires more than a communication interface between two programs. In LCF-style proof assistants [10] such as Isabelle, all proofs must be checked by a small trusted kernel. Thus, integrating external tools as unverified oracles is unsatisfactory: any error in the external tool or in

* Supported by the DFG grant GI 274/5-3 and the FWF project P22767-N13.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 152–167, 2011.

the integration code would compromise the overall soundness. Instead, the external tool must provide a certificate that can be checked by the proof assistant.

Our approach involves the following steps.

1. Generate the definition of a TRS $\mathcal{R}^f$ which corresponds to the function f.
2. Prove that termination of $\mathcal{R}^f$ indeed implies the termination goal for f.
3. Run the termination tool on $\mathcal{R}^f$ and obtain a certificate.
4. Replay the certificate using a formally verified checker.

While steps 1 and 3 are not hard, and the ground work for step 4 is already available in the IsaFoR library [17,19], which formalizes term rewriting and several termination techniques,[1] this paper is concerned with the missing piece, the reduction of termination proof obligations for HOL functions to the termination of a TRS. This is non-trivial, as the languages differ considerably. Termination of a TRS expresses the well-foundedness of a relation over terms, i.e., of type $(\mathit{term} \times \mathit{term})$ *set*, where *term*s are first-order terms. In contrast, the termination proof obligation for a HOL function states the well-foundedness of its call relation, which has the type $(\alpha \times \alpha)$ *set*, where α is the argument type of the function. In essence, we must move from a shallow embedding (the functional programming fragment of Isabelle/HOL) to a deep embedding (the formalization of term rewriting in IsaFoR).

The goal of this paper is to provide this formal relationship between termination of first-order HOL functions and termination of TRSs. More precisely, we develop a tactic that automatically reduces the termination proof obligation of a HOL function to the termination problem of a TRS. This allows us to use arbitrary termination tools for fully automated termination proofs inside Isabelle. Thus, powerful termination tools become available to the Isabelle user, while retaining the strong soundness guarantees of an LCF-style proof assistant. Since our approach is generic, it automatically benefits from future improvements to termination tools and the termination techniques within IsaFoR. Our implementation is available as part of IsaFoR.

Outline of this paper. We give a short introduction on term rewriting, HOL and HOL functions in §2. Then we show our main result in §3 on how to systematically discharge the termination proof obligation of a HOL function via proving termination of a TRS. In §4 we present some examples which show the strengths and limitations of our technique. How to extend our approach to support more HOL functions is discussed in §5. We conclude in §6.

2 Preliminaries

2.1 Higher-Order Logic

We consider classical HOL, which is based on simply-typed lambda-calculus, enriched with a simple form of ML-like polymorphism. Among its basic types are a type *bool* of truth values and a function space type constructor $\Rightarrow$ (where $\alpha \Rightarrow \beta$ denotes the type of total functions mapping values of type α to values of

[1] See http://cl-informatik.uibk.ac.at/software/ceta for a list of supported techniques.

type β). Sets are modeled by a type α *set*, which just abbreviates $\alpha \Rightarrow$ *bool*.

By an add-on tool, HOL supports algebraic datatypes, which includes the types *nat* (with constructors 0 and *Suc*) and *list* (with constructors [] and #).

Another add-on tool, the *function package* [13], completes the functional programming layer by allowing recursive function definitions, which are not covered by the primitives of the logic. Since it internally employs a well-founded recursion principle, it requires the user to prove well-foundedness of a certain relation, extracted automatically from the function definition (cf. §2.3). This proof obligation, by its construction, directly corresponds to the termination of the function being defined. It is the proof of this goal that we want to automate.

As opposed to functional programming languages, there is no operational semantics for HOL; the meaning of its expressions is instead given by a set-theoretic denotational semantics. As a consequence, there is no direct notion of evaluation or termination of an expression. Thus, when we informally say that we prove "termination of a HOL function," this simply means that we discharge the proof obligation produced by the function package.

2.2 Supported Fragment

Isabelle supports a wide spectrum of specifications, using various forms of inductive, coinductive and recursive definitions, as well as quantifiers and Hilbert's choice operator. Clearly, not all of them can be easily expressed using TRSs. Thus, we must limit ourselves to a subset which is sufficiently close to rewriting, and consider only algebraic datatypes, given by a set of constructors together with their types, and recursive functions, given by their defining equations with pattern matching. Additionally, we impose the following restrictions:

1. Functions and constructors must be first-order (no functions as arguments).
2. Patterns are constructor terms and must be linear and non-overlapping.
3. Patterns must be complete.
4. Expressions consist of variables, function applications, and case-expressions only. In particular, partial applications and λ-abstractions are excluded.

Linearity is always satisfied by function definitions that are accepted by Isabelle's function package, and pattern overlaps are eliminated automatically. For ease of presentation, we assume that there is no mutual recursion (f calls g and g calls f) and no nested recursion (arguments of a recursive call contain other recursive calls; they may of course contain calls to other defined functions).

Most of the above restrictions are not fundamental, and we discuss in §5 how some of them can be removed. Our chosen fragment of HOL rather represents a compromise between expressive power and a reasonably simple presentation and implementation of our reduction technique. Note that case-expressions encompass the simpler if-expressions, which can be seen as case-expressions on type *bool*. Isabelle's (non-recursive and monomorphic) let-expressions can simply be inlined or replaced by case-expressions if patterns are involved.

The functions *half* and *log* below (*log* computes the logarithm) illustrate our supported fragment and will be used as running examples throughout this paper.

$\mathit{half}\, 0 = 0$
$\mathit{half}\ (\mathit{Suc}\ 0) = 0$
$\mathit{half}\ (\mathit{Suc}\ (\mathit{Suc}\ n)) = \mathit{Suc}\ (\mathit{half}\ n)$
$\mathit{log}\ n = (\mathit{case}\ \mathit{half}\ n\ \mathit{of}\, 0 \Rightarrow 0 \mid \mathit{Suc}\ m \Rightarrow \mathit{Suc}\ (\mathit{log}\ (\mathit{Suc}\ m)))$

2.3 Function Definitions By Well-Founded Recursion

When the user writes a recursive definition, the function package analyzes the equations and extracts the recursive calls. This information is then used to synthesize the termination proof obligation.

Formally, we define the operation $\textsc{calls}_f$ that computes the set of calls to f inside an expression, each together with a condition under which it occurs.

- $\textsc{calls}_f(g\ e_1\ \dots\ e_k) \equiv \textsc{calls}_f(e_1) \cup \dots \cup \textsc{calls}_f(e_k)$ if g is a constructor or a defined function other than f,
- $\textsc{calls}_f(f\ e_1\ \dots\ e_n) \equiv \textsc{calls}_f(e_1) \cup \dots \cup \textsc{calls}_f(e_n) \cup \{(e_1, \dots, e_n, \mathit{True})\}$,
- $\textsc{calls}_f(x) \equiv \varnothing$ for all variables x, and
- $\textsc{calls}_f(\mathit{case}\ e\ \mathit{of}\, p_1 \Rightarrow e_1 \mid \dots \mid p_k \Rightarrow e_k) \equiv \textsc{calls}_f(e) \cup (\textsc{calls}_f(e_1) \wedge e = p_1) \cup \dots \cup (\textsc{calls}_f(e_k) \wedge e = p_k)$ where $\textsc{calls}_f(e_i) \wedge e = p_i$ is like $\textsc{calls}_f(e_i)$, but every $(t_1, \dots, t_m, \varphi) \in \textsc{calls}_f(e_i)$ is replaced by $(t_1, \dots, t_m, \varphi \wedge e = p_i)$.

The termination proof obligation requires us to exhibit a strongly normalizing relation $\succ$ such that for each defining equation $f\ p_1\ \dots\ p_n = e$ and each $(r_1, \dots, r_n, \phi) \in \textsc{calls}_f(e)$ we can prove $\phi \implies (p_1, \dots, p_n) \succ (r_1, \dots, r_n)$.

Consider for example the definition of *half*, where we have $\textsc{calls}_{\mathit{half}}(0) \equiv \varnothing$ and $\textsc{calls}_{\mathit{half}}(\mathit{Suc}\ (\mathit{half}\ n)) \equiv \{(n, \mathit{True})\}$. We obtain the following obligation.

1. $\mathit{SN}\ ?R$
2. $\forall n.\ (\mathit{Suc}\ (\mathit{Suc}\ n),\ n) \in\ ?R$

The variable $?R :: (\mathit{nat} \times \mathit{nat})\ \mathit{set}$ is a *schematic variable*, which can be instantiated during the proof, i.e., it can be seen as existentially quantified.

For *log*, we have $\textsc{calls}_{\mathit{log}}(\mathit{case}\ \mathit{half}\, n\ \mathit{of}\, 0 \Rightarrow 0 \mid \mathit{Suc}\, m \Rightarrow \mathit{Suc}\ (\mathit{log}\ (\mathit{Suc}\, m))) \equiv \{(\mathit{Suc}\ m,\ \mathit{half}\, n = \mathit{Suc}\ m)\}$, and the following proof obligation is produced.

1. $\mathit{SN}\ ?R$
2. $\forall n\ m.\ \mathit{half}\ n = \mathit{Suc}\ m \Longrightarrow (n,\ \mathit{Suc}\ m) \in\ ?R$

Two things should be noted here. First, the fact that the recursive call is guarded by a case-expression is reflected by a condition in the corresponding subgoal. Without this condition, which models the usual evaluation behavior of *case*, the goal would be unprovable. Second, the goal may refer to previously defined functions. To prove it, we must refer to properties of these functions, either through their definitions, or through other lemmas about them.

When the proof obligation is discharged, the function package can use the result to derive the recursive equations as theorems (previously, they were just conjectures—consider the recursive equation $f\ x = \mathit{Suc}\ (f\ x)$, which is inconsistent). Additionally, an induction rule is provided, which expresses “induction along the computation.” The induction rules for *half* and *log* are shown below.

$$P\ 0 \Longrightarrow P\ (Suc\ 0) \Longrightarrow (\forall n.\ P\ n \Longrightarrow P\ (Suc\ (Suc\ n))) \Longrightarrow \forall n.\ P\ n$$
$$(\forall n.\ (\forall m.\ half\ n = Suc\ m \Longrightarrow P\ (Suc\ m)) \Longrightarrow P\ n) \Longrightarrow \forall n.\ P\ n$$

2.4 IsaFoR - Term Rewriting Formalized in Isabelle/HOL

In the following, we assume that the reader is familiar with the basics of term rewriting [1]. Many notions and facts from rewriting have been formalized in the Isabelle library IsaFoR [19]. Before we can give the reduction from termination of HOL functions to termination of corresponding TRSs in §3, we need some more details on IsaFoR. Terms are represented straightforwardly by the datatype:

$$\textbf{datatype}\ (\alpha, \beta)\ term = Var\ \beta \mid Fun\ \alpha\ ((\alpha, \beta)\ term\ list)$$

The type variables α and β, which represent function and variable symbols, respectively, are always instantiated with the type *string* in our setting. Hence, we abbreviate $(string, string)\ term$ by *term* in the following. For example, the term $\mathsf{f}(\mathsf{x}, \mathsf{y})$ is represented by *Fun* "*f*" [*Var* "*x*", *Var* "*y*"]. A TRS is represented by a value of type $(term \times term)\ set$.

The semantics of a TRS is given by its rewrite relation $\to_{\mathcal{R}}$, defined by closing $\mathcal{R}$ under contexts and substitutions. Termination of $\mathcal{R}$ is formalized as $SN\ (\to_{\mathcal{R}})$.

IsaFoR formalizes many criteria commonly used in automated termination proofs. Ultimately, it contains an executable and terminating function

$$check\text{-}proof :: (term \times term)\ list \Rightarrow proof \Rightarrow bool$$

and a proof of the following soundness theorem:

Theorem 1 (Soundness of Check). $check\text{-}proof\ \mathcal{R}\ prf \Longrightarrow SN\ (\to_{\mathcal{R}})$

Here, *prf* is a certificate (i.e., a termination proof of $\mathcal{R}$) from some external source, encoded as a value of a suitable datatype, and $\mathcal{R}$ is the TRS under consideration.[2] Whenever *check-proof* returns *True* for some given TRS $\mathcal{R}$ and certificate *prf*, we have established (inside Isabelle) that *prf* is a valid termination proof for $\mathcal{R}$. Thus, we can prove termination of concrete TRSs inside Isabelle.

The technical details on the supported termination techniques and the structure of the certificate (i.e., the type *proof*) are orthogonal to our use of the check function, which only relies on Theorem 1.

2.5 Terminology and Notation

The layered nature of our setting requires that we carefully distinguish three levels of discourse. Primarily, there is higher-order logic (implemented in Isabelle/HOL), in which all mechanized reasoning takes place. The termination goals we ultimately want to solve are formulated on this level. Of course, the syntax of HOL consists of terms, but to distinguish them from the embedded

[2] To be executable, *check-proof* demands that $\mathcal{R}$ is given as a list of rules and not as a set. We ignore this difference, since it is irrelevant for this paper.

term language of term rewriting, we refer to them as *expressions*. They are uniformly written in *italics* and follow the conventions of the lambda-calculus (in particular, function application is denoted by juxtaposition). HOL equality is denoted by =. For example, the definition of *half* above is a HOL expression.

The second level is the "sub-language" of first-order terms, which is deeply embedded into HOL by the datatype *term*. When we speak of a *term*, we always refer to a value of that type, not an arbitrary HOL expression. While this embedding is simple and adequate, the concrete syntax with the *Fun* and *Var* constructors and string literals is rather unwieldy. Hence, for readability, we use sans-serif font to abbreviate the constructors and the quotes: Instead of *Var* "*v*" we write v, and instead of *Fun* "*f*" [...] we write $\mathsf{f}(\ldots)$, omitting the parentheses $()$ for nullary functions. This recovers the well-known concrete syntax of term rewriting, but we must keep in mind that the constructors and strings are still present, although they are not written as such.

Finally, we must relate the two languages with each other, and describe the proof procedures that derive the relevant properties. While the properties themselves can be stated in HOL for each concrete instance, the general schema cannot, as it must talk about "all HOL expressions." Thus, we use a meta-language as another level above HOL, in which we express the transformations and tactics. This level corresponds to our implementation (in ML). Functions of the meta-language are written in SMALL CAPITALS (e.g., $\textsc{calls}_f$), and variables of the meta-language, which typically range over arbitrary HOL expressions or patterns, are written e or p, possibly with annotations. For HOL expressions that are arguments of recursive calls we also use r. Equality of the meta-language is written $\equiv$ and denotes syntactic equality of HOL expressions. In particular, $e \equiv e'$ implies $e = e'$, since HOL's equality is reflexive.

Both embeddings are deep, that is, each level can talk about the syntax of the lower levels. As a simple example, the concept of a ground term can be defined as a recursive HOL function $\mathit{ground} :: \mathit{term} \Rightarrow \mathit{bool}$:

$$\begin{aligned} &\mathit{ground}\ (\mathit{Var}\ x) = \mathit{False} \\ &\mathit{ground}\ (\mathit{Fun}\ f\ ts) = (\forall t{\in}\mathit{set}(ts).\ \mathit{ground}\ t) \end{aligned}$$

Then we can immediately deduce that $\mathit{ground}\,(\mathsf{f}(\mathsf{x})) = \mathit{False}$, due to the presence of x. Note however that the similar-looking statement $\mathit{ground}\ (\mathsf{f}(x)) = \mathit{False}$ is not uniformly true. More precisely, its universal closure $\forall x.\ \mathit{ground}\,(\mathsf{f}(x)) = \mathit{False}$ does not hold, since we could instantiate x with the term c (i.e., *Fun* "*c*" [ ]). Thus, we must not confuse variables of the different levels. Obviously, we cannot quantify over a variable x, which is just the *Var* constructor applied to a string.

Similarly, the meta-language can talk about the syntax of HOL, as in the definition of $\textsc{calls}_f$, which is recursive over the structure of HOL expressions.

3 The Reduction to Rewriting

3.1 Encoding Expressions and Defining Equations

We define a straightforward encoding of HOL expressions as terms, denoted by the meta-level operation ENC. For case-free expressions, we turn variables into term variables and (curried) applications into applications on the term level:

$$\begin{aligned} \textsc{enc}(x) &\equiv \mathsf{x} \\ \textsc{enc}(f\ e_1\ \ldots\ e_n) &\equiv \mathsf{f}(\textsc{enc}(e_1), \ldots, \textsc{enc}(e_n)) \end{aligned}$$

Each case-expression is replaced by a new function symbol, for which we will include additional rules below. To simplify bookkeeping, we assume that each occurrence of a case-expression is annotated with a unique integer j.

$$\begin{aligned} &\textsc{enc}(\mathit{case}_j\ e\ \mathit{of}\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k) \\ &\quad \equiv \mathsf{case_j}(\textsc{enc}(e), \textsc{enc}(y_1), \ldots, \textsc{enc}(y_m)) \end{aligned}$$

where $y_1, \ldots, y_m$ are all variables that occur free in some e_i but not in p_i.

The operation RULES yields the rewrite rules for a function or case-expression. For a function f with defining equations $\ell_1 = r_1,\ \ldots,\ \ell_k = r_k$, they are

$$\textsc{rules}(f) \equiv \{\ \textsc{enc}(\ell_1) \to \textsc{enc}(r_1),\ \ldots,\ \textsc{enc}(\ell_k) \to \textsc{enc}(r_k)\ \}\,.$$

For the case-expression $\mathit{case}_j\ e\ \mathit{of}\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k$ we have

$$\begin{aligned} \textsc{rules}(\mathit{case}_j) \equiv \{\ &\mathsf{case_j}(\textsc{enc}(p_1), \textsc{enc}(y_1), \ldots, \textsc{enc}(y_m)) \to \textsc{enc}(e_1), \\ &\ldots, \\ &\mathsf{case_j}(\textsc{enc}(p_k)), \textsc{enc}(y_1), \ldots, \textsc{enc}(y_m)) \to \textsc{enc}(e_k)\ \}\,. \end{aligned}$$

We define the TRS for f as $\mathcal{R}^f = \textsc{rules}(f) \cup \bigcup_{g \in \mathcal{S}_f} \textsc{rules}(g)$ where $\mathcal{S}_f$ is the set of all functions that are used (directly or indirectly) by f. Our encoding is similar to the well known technique of unraveling which transforms conditional into unconditional TRSs [14,16].[3]

For example, $\mathcal{R}^{log}$ is defined as follows and completely contains $\mathcal{R}^{half}$.

$$\begin{aligned} \mathsf{half}(\mathsf{0}) &\to \mathsf{0} & \mathsf{log}(\mathsf{n}) &\to \mathsf{case_0}(\mathsf{half}(\mathsf{n})) \\ \mathsf{half}(\mathsf{Suc}(\mathsf{0})) &\to \mathsf{0} & \mathsf{case_0}(\mathsf{0}) &\to \mathsf{0} \\ \mathsf{half}(\mathsf{Suc}(\mathsf{Suc}(\mathsf{n}))) &\to \mathsf{Suc}(\mathsf{half}(\mathsf{n})) & \mathsf{case_0}(\mathsf{Suc}(\mathsf{m})) &\to \mathsf{Suc}(\mathsf{log}(\mathsf{Suc}(\mathsf{m}))) \end{aligned}$$

3.2 Embedding Functions

At this point, we have defined a translation, but we cannot reason about it in Isabelle, since ENC is only an extra-logical concept, defined on the meta-level. In fact, it is easy to see that it cannot be defined in HOL: If we had a HOL function *enc* satisfying $\mathit{enc}\ 0 = \mathsf{0}$ and $\mathit{enc}\ (\mathit{half}\ 0) = \mathsf{half}(\mathsf{0})$, we would immediately have a contradiction, since $\mathit{half}\ 0 = 0$, and $\mathsf{half}(\mathsf{0}) \neq \mathsf{0}$, but a function must always yield the same result on the same input.

In a typical reflection scenario, we would proceed by defining an interpretation for *term*. For example, if we were modeling the syntax of integer arithmetic expressions, then we could define a function $\mathit{eval} :: \mathit{term} \Rightarrow \mathit{int}$ (possibly also depending on a variable assignment) which interprets terms as integers. However,

[3] It would be possible to directly generate dependency pair problems. However, techniques like [18] and several termination tools rely on the notion of "minimal chains," which is not ensured by our approach.

in our setting, the result type of such a function is not fixed, as our terms represent HOL expressions of arbitrary types. Thus, the result type of *eval* would depend on the actual term it is applied to. This cannot be expressed in a logic without dependent types, which means we cannot use this approach here.

Instead, we take the opposite route: For all relevant types σ, we define a function $emb_\sigma :: \sigma \Rightarrow term$, mapping values of type σ to their canonical term representation.

Using Isabelle's type classes, we use a single overloaded function *emb*, which belongs to a type class *embeddable*. Concrete datatypes can be declared to be instances of this class by defining *emb*, usually by structural recursion w.r.t. the datatype. For example, here are the definitions for the types *nat* and *list*:

$$\begin{array}{ll} emb\ 0 = \mathsf{0} & emb\ [\,] = \mathsf{Nil} \\ emb\ (Suc\ n) = \mathsf{Suc}(emb\ n) & emb\ (x\ \#\ xs) = \mathsf{Cons}(emb\ x,\ emb\ xs) \end{array}$$

This form of definition is canonical for all algebraic datatypes, and suitable definitions of *emb* can be automatically generated for all user-defined datatypes, turning them into instances of the class *embeddable*. This is analogous to the instances generated automatically by Haskell's "deriving" statement. It is also possible to manually provide the definition of *emb* for other types if they behave like datatypes like the predefined type *bool* for the Booleans.

Note that by construction, the result of *emb* is always a constructor ground term. For a HOL expression e that consists only of datatype constructors, (e.g., $Suc\ (Suc\ 0)$), we have $emb\ e = \textsc{enc}(e)$. For other expressions this is not the case, e.g., $emb\ (half\ 0) = emb\ 0 = \mathsf{0}$, but $\textsc{enc}(half\ 0) \equiv \mathsf{half}(\mathsf{0})$.

To formulate our proofs, we need another encoding of expressions as terms: The operation GENC is a slight variant of ENC, which treats variables differently, mapping them to their embeddings instead of term variables.

$$\begin{aligned} \textsc{genc}(x) &\equiv emb\ x \\ \textsc{genc}(f\ e_1\ \dots\ e_n) &\equiv \mathsf{f}(\textsc{genc}(e_1), \dots, \textsc{genc}(e_n)) \\ \textsc{genc}(&case_j\ e\ of\ p_1 \Rightarrow e_1\ \mid \dots \mid p_k \Rightarrow e_k) \\ &\equiv \mathsf{case_j}(\textsc{genc}(e), \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) \end{aligned}$$

where $y_1, \dots, y_m$ are all variables that occur free in some e_i but not in p_i.

Hence, $\textsc{genc}(e)$ never contains term variables. However, it contains the same HOL variables as e. For example, $\textsc{genc}(half\ (Suc\ n)) \equiv \mathsf{half}(\mathsf{Suc}(emb\ n))$.

3.3 Rewrite Lemmas

The definitions of $\mathcal{R}^{half}$ and $\mathcal{R}^{log}$ above are straightforward, but reasoning with them is clumsy and low-level: To establish a single rewrite step, we must extract the correct rule (that is, prove that it is in the set $\mathcal{R}^{half}$ or $\mathcal{R}^{log}$), invoke closure under substitution, and construct the correct substitution explicitly as a function of type $string \Rightarrow term$.

To avoid such repetitive reasoning, we automatically derive an individual lemma for each rewrite rule. From the definition of $\mathcal{R}^{half}$, we obtain the following rules, which we call *rewrite lemmas*:

$$\mathsf{half}(\mathsf{0}) \to_{\mathcal{R}^{\mathit{half}}} \mathsf{0} \qquad \mathsf{half}(\mathsf{Suc}(\mathsf{0})) \to_{\mathcal{R}^{\mathit{half}}} \mathsf{0}$$
$$\forall t.\ \mathsf{half}(\mathsf{Suc}(\mathsf{Suc}(t))) \to_{\mathcal{R}^{\mathit{half}}} \mathsf{Suc}(\mathsf{half}(t))$$

Note that the term variable n in the last rule has been turned into a universally-quantified HOL variable by applying the "generic substitution" $\{\mathsf{n} \mapsto t\}$. The advantage of this format is that applying a rewrite rule merely involves instantiating a universal quantifier, for which we can use the matching facilities of Isabelle. In particular, we can instantiate t with *emb* n, which in general results in a rewrite lemma of the form $\text{GENC}(f\ p_1\ \ldots\ p_n) \to_{\mathcal{R}} \text{GENC}(e)$ for a defining equation $f\ p_1\ \ldots\ p_n = e$.

3.4 The Simulation Property

The following property connects our generated TRSs with HOL expressions.

Definition 2 (Simulation Property). *For every expression e and $\mathcal{R} = \bigcup\{\mathcal{R}^f \mid f$ occurs in $e\}$, the* simulation property for *e is the statement*

$$\text{GENC}(e) \to^*_{\mathcal{R}} \mathit{emb}\ e.$$

As we cannot quantify over all HOL expressions within HOL itself, we cannot formalize that the simulation property holds for all e.

However, we will devise a tactic that derives this property for any given concrete expression. The basic building blocks of such proofs are lemmas of the following form, which are derived for each function symbol and can be composed to show the simulation property for a given expression.

Definition 3 (Simulation Lemma). *The* simulation lemma *for a function f of arity n is the statement*

$$\forall x_1 \ldots x_n.\ \mathsf{f}(\mathit{emb}\ x_1, \ldots, \mathit{emb}\ x_n) \to^*_{\mathcal{R}^f} \mathit{emb}\ (f\ x_1\ \ldots\ x_n)\ .$$

E.g., the simulation lemma for *half* is $\forall n.\ \mathsf{half}(\mathit{emb}\ n) \to^*_{\mathcal{R}^{\mathit{half}}} \mathit{emb}\ (\mathit{half}\ n)$.

The lemma claims that the rules that we produced for f can indeed be used to reduce a function application to the (embedding of) the value of the function. Of course, this way of saying "$\mathcal{R}^f$ computes f" admits the possibility that there are other $\mathcal{R}^f$-reductions that lead to different normal forms or that do not terminate, since we are not requiring confluence or strong normalization. But this form of simulation lemma is sufficient for our purpose.

We show in §3.6 how simulation lemmas are proved automatically.

3.5 Reduction of Termination Goals

After having proved termination of $\mathcal{R}^f$ using a termination tool in combination with IsaFoR and Theorem 1, we now show how to use this result to solve the termination goal for the HOL function f. Recall from §2.3 that we must exhibit a strongly normalizing relation $\succ$ such that $\phi \implies (p_1, \ldots, p_n) \succ (r_1, \ldots, r_n)$ for all $(r_1, \ldots, r_n, \phi) \in \text{CALLS}_f(e)$ for each defining equation $f\ p_1\ \ldots\ p_n = e$.

To this end, we first define $\rightsquigarrow$ as $\to_{\mathcal{R}^f} \cup \rhd$ where $\rhd$ is the strict subterm relation. The addition of $\rhd$ is required to strip off constructors and non-recursive function applications that are wrapped around recursive calls in right-hand sides of $\mathcal{R}^f$. Since $\to_{\mathcal{R}^f}$ is strongly normalizing and closed under contexts, also $\rightsquigarrow$ is strongly normalizing. This allows us to finally choose $\succ$ as the following relation.

$$(x_1, \ldots, x_n) \succ (y_1, \ldots, y_n) \text{ iff } \mathsf{f}(\mathit{emb}\ x_1, \ldots, \mathit{emb}\ x_n) \rightsquigarrow^+ \mathsf{f}(\mathit{emb}\ y_1, \ldots, \mathit{emb}\ y_n)$$

It remains to show that the arguments of recursive calls decrease w.r.t. $\succ$. That is, for each recursive call we have a goal of the form

$$\phi \implies \mathsf{f}(\mathit{emb}\, p_1, \ldots, \mathit{emb}\, p_n) \rightsquigarrow^+ \mathsf{f}(\mathit{emb}\, r_1, \ldots, \mathit{emb}\, r_n)$$

where $f\ p_1\ \ldots\ p_n = e$ is a defining equation of f and $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e)$. In the following, we illustrate the automated proof of this goal.

Note that since the p_i's are patterns, we have $\mathit{emb}\ p_i = \textsc{genc}(p_i)$, and hence

$$\begin{aligned} & \mathsf{f}(\mathit{emb}\ p_1, \ldots, \mathit{emb}\ p_n) & \\ = {} & \mathsf{f}(\textsc{genc}(p_1), \ldots, \textsc{genc}(p_n)) & & (p_i \text{ are patterns}) \\ \equiv {} & \textsc{genc}(f\ p_1\ \ldots\ p_n) & & (\text{definition of } \textsc{genc}) \\ \to_{\mathcal{R}^f} {} & \textsc{genc}(e) & & (\text{rewrite lemma}) \end{aligned}$$

Thus, it remains to construct a sequence $\textsc{genc}(e) \rightsquigarrow^* \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n)$, which reduces the right-hand side of the definition to a particular recursive call, eliminating any surrounding context. We proceed recursively over e.

- If $e \equiv g\ e_1\ \ldots\ e_m$ for a constructor g or a defined function symbol $g \not\equiv f$, then $(r_1, \ldots, r_n, \phi) \in \textsc{calls}(e_i)$ for some particular i. Hence, we have

$$\begin{aligned} & \textsc{genc}(e) & \\ \equiv {} & \mathsf{g}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_m)) & & (\text{definition of } \textsc{genc}) \\ \rhd {} & \textsc{genc}(e_i) & & (\text{definition of } \rhd) \\ \rightsquigarrow^* {} & \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n) & & (\text{apply tactic recursively}) \end{aligned}$$

- If $e \equiv f\ e_1\ \ldots\ e_n$ then (since we excluded nested recursion) we have $e_i = r_i$ for all i. Hence, we have

$$\begin{aligned} & \textsc{genc}(e) & \\ \equiv {} & \mathsf{f}(\textsc{genc}(r_1), \ldots, \textsc{genc}(r_n)) & & (\text{definition of } \textsc{genc}) \\ \to_{\mathcal{R}^f}^* {} & \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n) & & (\text{simulation property}) \end{aligned}$$

- If $e \equiv\ \mathit{case}_j\ e_0\ \mathit{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ then we distinguish where the recursive call is located. If $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e_0)$, then we have

$$\begin{aligned} & \textsc{genc}(e) & \\ \equiv {} & \mathsf{case_j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & & (\text{definition of } \textsc{genc}) \\ \rhd {} & \textsc{genc}(e_0) & & (\text{definition of } \rhd) \\ \rightsquigarrow^* {} & \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n) & & (\text{apply tactic recursively}) \end{aligned}$$

  Otherwise, $\phi \equiv (\chi \wedge e_0 = p_i)$ for some χ and $1 \leqslant i \leqslant k$, and $(r_1, \ldots, r_n, \chi) \in \textsc{calls}(e_i)$. We may therefore use the assumption $e_0 = p_i$ and proceed with

$$\begin{aligned}
&\textsc{genc}(e) \\
\equiv\ & \mathsf{case}_{\mathsf{j}}(\textsc{genc}(e_0), \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(definition of } \textsc{genc}\text{)} \\
\to^*_{\mathcal{R}^f}\ & \mathsf{case}_{\mathsf{j}}(\mathit{emb}\ e_0, \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(simulation property)} \\
=\ & \mathsf{case}_{\mathsf{j}}(\mathit{emb}\ p_i, \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(assumption } e_0 = p_i\text{)} \\
=\ & \mathsf{case}_{\mathsf{j}}(\textsc{genc}(p_i), \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(since } p_i \text{ is a pattern)} \\
\to_{\mathcal{R}^f}\ & \textsc{genc}(e_i) && \text{(rewrite lemma)} \\
\leadsto^*\ & \mathsf{f}(\mathit{emb}\ r_1, \dots, \mathit{emb}\ r_n) && \text{(apply tactic recursively)}
\end{aligned}$$

3.6 Proof of the Simulation Property

We have seen that for the reduction of termination goals it is essential to use the simulation property $\textsc{genc}(e) \to^*_{\mathcal{R}^f} \mathit{emb}\ e$ for expressions e that occur below recursive calls or within conditions that guard a recursive call. Below, we show how this property is derived for an individual expression, assuming that we already have simulation lemmas for all functions that occur in it. We again proceed recursively over e.

- If e is a HOL variable x then $\textsc{genc}(e) \equiv \textsc{genc}(x) \equiv \mathit{emb}\ x \equiv \mathit{emb}\ e$ and thus, the result follows by reflexivity of $\to^*_{\mathcal{R}^f}$.
- If $e \equiv g\ e_1\ \dots\ e_k$ for a function symbol g then

$$\begin{aligned}
&\textsc{genc}(e) \\
\equiv\ & \mathsf{g}(\textsc{genc}(e_1), \dots, \textsc{genc}(e_k)) && \text{(definition of } \textsc{genc}\text{)} \\
\to^*_{\mathcal{R}^f}\ & \mathsf{g}(\mathit{emb}\ e_1, \dots, \mathit{emb}\ e_k) && \text{(apply tactic recursively)} \\
\to^*_{\mathcal{R}^f}\ & \mathit{emb}\ (g\ e_1\ \dots\ e_k) && \text{(simulation lemma for } g\text{)} \\
\equiv\ & \mathit{emb}\ e
\end{aligned}$$

- If $e \equiv \mathit{case}_j\ e_0\ \mathit{of}\ p_1 \Rightarrow e_1 \mid \dots \mid p_k \Rightarrow e_k$ then we construct the following rewrite sequence:

$$\begin{aligned}
&\textsc{genc}(e) \\
\equiv\ & \mathsf{case}_{\mathsf{j}}(\textsc{genc}(e_0), \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(definition of } \textsc{genc}\text{)} \\
\to^*_{\mathcal{R}^f}\ & \mathsf{case}_{\mathsf{j}}(\mathit{emb}\ e_0, \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(apply tactic recursively)}
\end{aligned}$$

  Now we apply a case analysis on e_0, which must be equal (in HOL, not syntactically) to one of the patterns. In each particular case we may assume $e_0 = p_i$. Then we continue:

$$\begin{aligned}
&\mathsf{case}_{\mathsf{j}}(\mathit{emb}\ e_0, \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) \\
=\ & \mathsf{case}_{\mathsf{j}}(\mathit{emb}\ p_i, \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(assumption } e_0 = p_i\text{)} \\
=\ & \mathsf{case}_{\mathsf{j}}(\textsc{genc}(p_i), \textsc{genc}(y_1), \dots, \textsc{genc}(y_m)) && \text{(since } p_i \text{ is a pattern)} \\
\to_{\mathcal{R}^f}\ & \textsc{genc}(e_i) && \text{(rewrite lemma)} \\
\to^*_{\mathcal{R}^f}\ & \mathit{emb}\ e_i && \text{(apply tactic recursively)} \\
=\ & \mathit{emb}\ e && \text{(assumption } e_0 = p_i\text{)}
\end{aligned}$$

The tactic above assumes that simulation lemmas for all functions in e are already present. Note the simulation lemma is trivial to prove if f is a constructor, since $\mathsf{f}(emb\ x_1, \ldots, emb\ x_n) = emb\ (f\ x_1\ \ldots\ x_n)$ by definition of *emb*.

For defined symbols of non-recursive functions the simulation lemmas are derived by unfolding the definition of the function and applying the tactic above. Thus, simulation lemmas are proved bottom-up in the order of function dependencies. When a function is recursive, the proof of its simulation lemma proceeds by induction using the induction principle from the function definition.

Example 4. We show how the simulation lemma for *log* is proved, assuming that the simulation lemmas for 0, *Suc*, and *half* are already available.

So our goal is to show $\mathsf{log}(emb\ n) \to^*_{\mathcal{R}^{log}} emb\ (log\ n)$ for any $n :: nat$. We apply the induction rule of *log* and obtain the following induction hypothesis.

$$\forall m.\ half\ n = Suc\ m \implies \mathsf{log}(emb\ (Suc\ m)) \to^*_{\mathcal{R}^{log}} emb\ (log\ (Suc\ m))$$

Let c abbreviate $case\ half\ n\ of\ 0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m))$. Then

$$\begin{aligned} & \mathsf{log}(emb\ n) & \\ \to_{\mathcal{R}^{log}}\ & \mathsf{case_0}(\mathsf{half}(emb\ n)) & \text{(rewrite lemma)} \\ \to^*_{\mathcal{R}^{log}}\ & \mathsf{case_0}(emb\ (half\ n)) & \text{(simulation lemma of } half\text{)} \end{aligned}$$

We continue by case analysis on *half n*. We only present the more interesting case $half\ n = Suc\ m$ (the other case $half\ n = 0$ is similar):

$$\begin{aligned} & \mathsf{case_0}(emb\ (half\ n)) & \\ =\ & \mathsf{case_0}(emb\ (Suc\ m)) & \text{(assumption } half\ n = Suc\ m\text{)} \\ =\ & \mathsf{case_0}(\mathsf{Suc}(emb\ m)) & \text{(def. of } emb\text{)} \\ \to_{\mathcal{R}^{log}}\ & \mathsf{Suc}(\mathsf{log}(\mathsf{Suc}(emb\ m))) & \text{(rewrite lemma)} \\ \to^*_{\mathcal{R}^{log}}\ & \mathsf{Suc}(\mathsf{log}(emb\ (Suc\ m))) & \text{(simulation lemma of } Suc\text{)} \\ \to^*_{\mathcal{R}^{log}}\ & \mathsf{Suc}(emb\ (log\ (Suc\ m))) & \text{(induction hypothesis)} \\ \to^*_{\mathcal{R}^{log}}\ & emb\ (Suc\ (log\ (Suc\ m))) & \text{(simulation lemma of } Suc\text{)} \\ =\ & emb\ c & \text{(assumption } half\ n = Suc\ m\text{)} \\ =\ & emb\ (log\ n) & \text{(def. of } log\text{)} \end{aligned}$$

4 Examples

We show some characteristic examples that illustrate the strengths and weaknesses of our approach. Each example is representative for several similar ones that occur in the Isabelle distribution.

Example 5. Consider binary trees defined by the type

datatype $tree = E \mid N\ tree\ nat\ tree$

and a function *remdups* that removes duplicates from a tree. The function is defined by the following equations (the auxiliary function *del* removes all occurrences of an element from a tree; we omit its straightforward definition here):

$$\begin{aligned}&\mathit{remdups}\ E = E\\ &\mathit{remdups}\ (N\ l\ x\ r) = N\ (\mathit{remdups}\ (\mathit{del}\ x\ l))\ x\ (\mathit{remdups}\ (\mathit{del}\ x\ r))\end{aligned}$$

The termination argument for *remdups* relies on a property of *del*: the result of *del* is smaller than its argument. In Isabelle, the user must manually state and prove (by induction) the lemma $\mathit{size}\ (\mathit{del}\ x\ t) \leq \mathit{size}\ t$, before termination can be shown. Here, *size* is an overloaded function generated automatically for every algebraic datatype.

For a termination tool, termination of the related TRS is easily proved using standard techniques, eliminating the need for finding and proving the lemma.

Example 6. The following function (originally due to Boyer and Moore [4]) normalizes conditional expressions consisting of atoms (AT) and if-expressions (IF).

$$\begin{aligned}&\mathit{norm}\ (AT\ a) = AT\ a\\ &\mathit{norm}\ (IF\ (AT\ a)\ y\ z) = IF\ (AT\ a)\ (\mathit{norm}\ y)\ (\mathit{norm}\ z)\\ &\mathit{norm}\ (IF\ (IF\ u\ v\ w)\ y\ z) = \mathit{norm}\ (IF\ u\ (IF\ v\ y\ z)\ (IF\ w\ y\ z))\end{aligned}$$

Isabelle's standard size measure is not sufficient to prove termination of *norm*, and a custom measure function must be specified by the user. Using a termination tool, the proof is fully automatic and no measure function is required.

Example 7. The Isabelle distribution contains the following implementation of the merge sort algorithm (transformed into non-overlapping rules internally):

$$\begin{aligned}&\mathit{msort}\ [\,] = [\,]\\ &\mathit{msort}\ [x] = [x]\\ &\mathit{msort}\ xs = \mathit{merge}\ (\mathit{msort}\ (\mathit{take}\ (\mathit{length}\ xs\ \mathit{div}\ 2)\ xs))\ (\mathit{msort}\ (\mathit{drop}\\ &(\mathit{length}\ xs\ \mathit{div}\ 2)\ xs))\end{aligned}$$

The situation is similar to Example 5, as we must know how *take* and *drop* affect the length of the list. However, in this case, Isabelle's list theory already provides the necessary lemmas, e.g., $\mathit{length}\ (\mathit{take}\ n\ xs) = \mathit{min}\ n\ (\mathit{length}\ xs)$. Together with the built-in arithmetic decision procedures (which know about *div* and *min*), the termination proof works fully automatically.

For termination tools, the proof is a bit more challenging and requires techniques that are not yet formalized in IsaFoR (in particular, the technique of *rewriting dependency pairs* [8]). Thus, our connection to termination tools cannot handle *msort* yet. However, when this technique is added to IsaFoR in the future, no change will be required in our implementation to benefit from it.

These examples show the main strength of our reduction to rewriting: absolutely no user input in the form of lemmas or measure functions is required. On the other hand, Isabelle's ability to pick up previously established results can make the built-in termination prover surprisingly strong in the presence of a good library, as the *msort* example shows. Even though that example can be solved by termination tools (and only the formalization lags behind), it shows an intrinsic weakness of the approach, since existing facts are not used and must be rediscovered by the termination tool if necessary.

5 Extensions

In this section, we reconsider the restrictions imposed in §2.2.

Nested Recursion. So far, we excluded nested recursion like $f\ (Suc\, n) = f\ (f\ n)$. The problem is that to prove termination of f we need its simulation lemma to reduce the inner call in the proof of the outer call, cf. §3.5. But proving the simulation lemma uses the induction rule of f, which in turn requires termination.

To solve this problem, we can use the partial induction rule that is generated by the function package even before a termination proof [13]. This rule, which is similar to the one used previously, contains extra domain conditions of the form $dom_f\ x$. It allows us to derive the restricted simulation lemma $dom_f\ n \implies \mathsf{f}(emb\ n) \to^*_{\mathcal{R}^f} emb\ (f\ n)$. In the termination proof obligation for the outer recursive call, we may assume this domain condition for the inner call (a convenience provided by the function package), so that this restricted form of simulation lemma suffices. Hence, dealing with nested recursion simply requires a certain amount of additional bookkeeping.

Underspecification. So far, we require functions to be completely defined, i.e., no cases are missing in left-hand sides or case-expressions. However, $head\ (x \# xs) = x$ is a common definition. It is internally completed by $head\ [\,] = undefined$ in Isabelle, where $undefined :: \alpha$ is an arbitrary but unknown value of type α.

For such functions, we cannot derive the simulation lemma, since this would require $\mathsf{head}(\mathsf{Nil})$ to be equal to *emb undefined*, which is an unknown term of the form $\mathsf{Suc}^k(\mathsf{0})$. The obvious idea of adding the rule $\mathsf{head}(\mathsf{Nil}) \to \mathsf{undefined}$ to the TRS does not work, since $\mathsf{undefined}$ cannot be equal to *emb undefined*.

We can solve the problem by using fresh variables for unspecified cases, e.g., by adding the rule $\mathsf{head}(\mathsf{Nil}) \to \mathsf{x}$. Then, the simulation lemma holds. However, the resulting TRS is no longer terminating. This new problem can be solved by using a variant of innermost rewriting, which would require support by IsaFoR as well as the termination tool.

Non-Representable Types and Polymorphism. Clearly, our embedding is limited to types that admit a term representation. This excludes uncountable types such as real numbers and most function types. However, even if such types occur in HOL functions, they may not be relevant for termination. Then, we can simply map all such values to a fixed constant by defining, e.g., $emb\ (r :: real) = \mathsf{real}$. Hence, the simulation lemmas for functions returning real numbers are trivial to prove. Furthermore, a termination proof that does not rely on these values works without problems. Like for underspecified functions, the generated TRS no longer models the original function completely, but is only an abstraction that is sufficient to prove termination.

A similar issue arises with polymorphic functions: To handle a function of type α *list* $\Rightarrow$ α *list* we need a definition of *emb* on type α. Mapping values of type α to a constant is unproblematic, since the definition is irrelevant for the proof. However, a class instance α :: *embeddable* would violate the type class discipline. This can be solved by either replacing the use of type classes by explicit dictionary constructions (where emb_{list} would take the embedding function to use for the list elements as an argument), or by restricting α to class *embeddable*. Since the type class does not carry any axioms, the system allows us to remove the class constraint from the final theorem, so no generality is lost.

Higher-Order Functions. Higher-order functions pose new difficulties. First, we cannot hope to define *emb* on function types. In particular, this means that we cannot even state the simulation lemma for a function like *map*. Second, the termination conditions for functions with higher-order recursion depend on user-provided congruence rules of a certain format [13]. These congruence rules then influence the form of the premise ϕ in the termination condition.

A partial solution could be to create a first-order function map_f for each invocation of *map* on a concrete function f. Commonly used combinators like *map*, *filter* and *fold* could be supported in this way.

6 Conclusion

We have presented a generic approach to discharge termination goals of HOL functions by proving termination of a corresponding generated TRS. Hence, where before a manual termination proof might have been required, now external termination tools can be used. Since our approach is not tied to any particular termination proof technique, its power scales up as the capabilities of termination tools increase and more techniques are formalized in IsaFoR.

A complete prototype of our implementation is available in the IsaFoR/CeTA distribution (version 1.18, http://cl-informatik.uibk.ac.at/software/ceta), which also includes usage examples. It remains as future work to extend our approach to a larger class of HOL functions. Moreover, the implementation has to be more smoothly embedded into the Isabelle system such that a user can easily access the provided functionality. The general approach is not limited to Isabelle, and could be ported to other theorem provers like Coq, which has similar recursive definition facilities (e.g., [2]) and rewriting libraries similar to IsaFoR [3,6].

Acknowledgment. Jasmin Blanchette gave helpful feedback on a draft of this paper.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1999)
2. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 114–129. Springer, Heidelberg (2006)

3. Blanqui, F., Koprowski, A.: CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Math. Struct. Comp. Science (2011) (to appear)
4. Boyer, R.S., Moore, J S.: A Computational Logic. Academic Press, London (1979)
5. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer, Heidelberg (2007)
6. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Certification of automated termination proofs. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 148–162. Springer, Heidelberg (2007)
7. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. J. Autom. Reasoning 40(2-3), 195–220 (2008)
8. Giesl, J., Arts, T.: Verification of Erlang processes by dependency pairs. Appl. Algebr. Eng. Comm 12(1,2), 39–72 (2001)
9. 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)
10. Gordon, M.: From LCF to HOL: A short history. In: Proof, Language, and Interaction, pp. 169–185. MIT Press, Cambridge (2000)
11. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)
12. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
13. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning 44(4), 303–336 (2010)
14. Marchiori, M.: Logic programs as term rewriting systems. In: Rodríguez-Artalejo, M., Levi, G. (eds.) ALP 1994. LNCS, vol. 850, pp. 223–241. Springer, Heidelberg (1994)
15. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Ohlebusch, E.: Termination of logic programs: Transformational methods revisited. Appl. Algebr. Eng. Comm. 12(1-2), 73–116 (2001)
17. Sternagel, C.: Automatic Certification of Termination Proofs. PhD thesis, Institut für Informatik, Universität Innsbruck, Austria (2010)
18. Sternagel, C., Thiemann, R.: Certified subterm criterion and certified usable rules. In: Proc. RTA 2010, LIPIcs, vol. 6, pp. 325–340 (2010)
19. 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)
20. Zantema, H.: Termination of term rewriting by semantic labelling. Fundamenta Informaticae 24, 89–105 (1995)

Validating QBF Validity in HOL4

Ramana Kumar and Tjark Weber[⋆]

Computer Laboratory, University of Cambridge, UK
{rk436,tw333}@cam.ac.uk

Abstract. The Quantified Boolean Formulae (QBF) solver Squolem can generate certificates of validity, based on Skolem functions. We present independent checking of these certificates in the HOL4 theorem prover. This enables HOL4 users to benefit from Squolem's automation for valid QBF problems. Detailed performance data shows that LCF-style checking of validity certificates is often (but not always) feasible even for large QBF instances. Additionally, our work provides high correctness assurances for Squolem's claims of validity and uncovered a soundness bug in a previous version of its certificate validator QBV.

1 Introduction

Quantified Boolean Formulae (QBF) extend propositional logic with universal and existential quantification over Boolean variables. QBF have numerous applications in adversarial planning and formal verification [1,2,3]; for instance, they enable succinct encodings of bounded and unbounded model checking problems [4]. As a simple example, consider the formula

$$\forall x\, \exists y\, \exists z.\ (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z), \quad (1)$$

which says for all x there is a y that implies $x \oplus y$.

Deciding the validity of QBF is an extension of the well-known Boolean satisfiability problem (SAT). A propositional formula ϕ in Boolean variables $x_1, \ldots, x_n$ is satisfiable if and only if the QBF $\exists x_1 \ldots \exists x_n.\, \phi$ is valid. However, SAT is merely NP-complete, while QBF is the canonical PSPACE-complete problem [5]. Satisfiable propositional formulae have short certificates—namely, their satisfying assignments—that can be validated in polynomial time. For valid QBF, there is no known way to even specify a solution succinctly.

Nevertheless, certain QBF solvers can produce certificates for their answers that can be checked independently [6]. Squolem is a state-of-the-art QBF solver that generates certificates for valid formulae in a unified format based on finitary Boolean Skolem functions [7].

In this paper, we present independent checking of these certificates in the HOL4 [8] theorem prover. HOL4 is a popular interactive theorem prover for higher-order logic [9]. It is based on a small LCF-style [10,11] kernel that provides an abstract data type of theorems, equipped with a fixed set of constructor functions. Each function corresponds to an axiom schema or inference rule

[⋆] This work was supported by the British EPSRC under grant EP/F067909/1.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 168–183, 2011.

of higher-order logic. Derived rules that are not provided by this kernel must be implemented by composing existing rules. This provides high correctness assurances: derived rules cannot produce inconsistent theorems, as long as the theorem data type itself is implemented correctly. On the other hand, it makes an efficient implementation of derived rules challenging.

Our work is motivated primarily by a desire for increased automation in interactive theorem proving. Systems like Coq [12], HOL4, Isabelle [13] and PVS [14] can greatly benefit from the reasoning power of automated tools. This has been demonstrated numerous times, e.g., by integrations of SAT [15] and SMT solvers [16,17] as well as automated first-order provers [18,19,20]. We envision that HOL4 users might invoke Squolem directly to solve suitable proof obligations, but also that our integration might serve as a foundation, on top of which decision procedures for richer logics can be implemented through QBF encodings. Since the results are checked by HOL4's inference kernel, no trust needs to be put in the QBF solver.

An additional motivation arises from the fact that correctness of QBF solvers is hard to establish. QBF solvers are complex software tools that employ sophisticated heuristics and optimizations [21,22]. Different solvers may disagree on the status of individual benchmarks. QBF-Eval competitions until 2006 resolved disagreements by majority vote [23]. This rather unsatisfactory approach (which has been replaced by certificate checking in recent years) confirms the importance of QBF benchmark certification. HOL4's inference kernel has been scrutinized by dozens of researchers for over two decades. By using HOL4 as an independent checker, we obtain high correctness assurances for Squolem's results.

We review related work in Section 2, before introducing relevant background material in Section 3. Our main contribution, an efficient LCF-style implementation of certificate checking for valid QBF, is presented in Section 4. We evaluate our implementation in Section 5, and conclude in Section 6.

2 Related Work

This paper complements previous work on LCF-style checking for QBF certificates of *in*validity. In [24], an algorithm was presented that, given a QBF ϕ, obtains a HOL4 theorem $\vdash \neg\phi$ from a Squolem-generated certificate of invalidity. Here we present an algorithm to obtain a HOL4 theorem $\vdash \phi$ from a certificate of validity. Certificates of validity and invalidity for QBF are quite different. The latter employ Q-resolution [25], a refutation-complete inference rule that extends propositional resolution to quantified Boolean logic. The former (as considered here) are based on Skolem functions (see Section 3). In principle, one could establish validity of ϕ from invalidity of $\neg\phi$. However, this approach is not feasible in practice: current QBF solvers usually find inverted valid instances considerably harder and often time out [7]. Therefore, it is a practical necessity to support certificates of validity directly. We do not know whether inverted invalid instances become easier to solve as validity problems. It would be interesting to understand when one approach is superior to the other, based on the shape of a formula.

Other related work concerns the integration of automated solvers with LCF-style theorem provers, and certificate checking for QBF solvers. Integrations have been proposed, e.g., for first-order provers [18,19,20], for model checkers [26], for computer algebra systems [27,28,29], and more recently for SMT solvers [16,17]. We use a HOL4 integration of SAT solvers [15] in this work.

Narizzano et al. [6] give an overview of certificate checking for QBF solvers. Squolem's certificates show competitive performance, and they are relatively simple. Unsurprisingly, stand-alone proof checkers for QBF are typically much more efficient than the LCF-style proof checker presented here. However, they arguably do not provide the same degree of trustworthiness as the HOL4 kernel.

3 Background and Theory

We now introduce relevant terminology and describe the QBF certificate format (Section 3.3) as well as HOL4's inference calculus (Section 3.4). Propositional logic is presupposed.

3.1 Quantified Boolean Formulae

We assume an infinite set of Boolean variables. A *literal* is a possibly negated Boolean variable. When l is a literal, we write $\underline{l}$ to denote its variable. A *clause* is a disjunction of literals. We say that a propositional formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

Definition 1 (Quantified Boolean Formula). *A* Quantified Boolean Formula (QBF) *is of the form*

$$Q_1 x_1 \ldots Q_n x_n . \phi,$$

where $n \geq 0$, each x_i is a Boolean variable, each Q_i is either $\forall$ or $\exists$, and ϕ is a propositional formula in CNF.

$Q_1 x_1 \ldots Q_n x_n$ is called the *quantifier prefix*, and ϕ is called the *matrix*. Without loss of generality, we consider QBF in this *prenex form* only. Any formula involving only propositional connectives and quantifiers over Boolean variables can be transformed into prenex form through straightforward syntactic manipulations. (We have not yet implemented such a transformation in HOL4.) The *innermost* variable of the above QBF is x_n.

The QDIMACS format [30] is the standard input format of QBF solvers. It provides a textual means of encoding QBF in prenex form. It is a backward-compatible extension of the DIMACS format [31], the standard input format of SAT solvers. We have implemented a translation from (the QBF subset of) HOL4 terms to QDIMACS, and a simple recursive-descent parser for QDIMACS files that returns the corresponding QBF as a HOL4 term (see Section 3.4).

The QDIMACS format imposes further restrictions: all variables x_i must be distinct, all variables must appear in the matrix, and the innermost quantifier must be existential (i.e., $Q_n = \exists$). We further require all variables that appear in the matrix to be bound by some quantifier, i.e., we consider *closed* QBF only. This is to avoid confusion: in the QDIMACS format, free variables have existential semantics (to retain backward compatibility with DIMACS), while in HOL4, free variables in theorems have universal semantics (to permit instantiation). If a QBF has free variables, we consider its existential closure instead.

$\mathbb{B} = \{\top, \bot\}$ denotes the set of truth values. The semantics of closed QBF is defined recursively: $[\![\forall x.\, \phi]\!] = [\![\phi[x \mapsto \top] \wedge \phi[x \mapsto \bot]]\!]$, and similarly $[\![\exists x.\, \phi]\!] = [\![\phi[x \mapsto \top] \vee \phi[x \mapsto \bot]]\!]$. (Here $\phi[x \mapsto y]$ denotes substitution of y for all free occurrences of x in ϕ.) A QBF is called *valid* if its semantics is $\top$ (i.e., true).

3.2 Skolem Functions and Models

QBF of interest typically contain several dozen or even hundreds of quantifiers. A naive recursive computation of their semantics, which would be exponential in the number of quantifiers, is not feasible. Therefore, QBF solvers implement different algorithms. The certificates of validity that we consider here are based on finitary Boolean Skolem functions [7].

Definition 2 (Model). *A* model *of the QBF $Q_1 x_1 \ldots Q_n x_n.\, \phi$ maps each existentially quantified variable x_k to a function $f_k \colon \mathbb{B}^{k-1} \to \mathbb{B}$.*

A model provides a witness function for every existentially quantified variable. We identify each witness function f_k with a propositional formula in $k-1$ variables. Because Skolemization preserves satisfiability, we have

Theorem 1. *A QBF $Q_1 x_1 \ldots Q_n x_n.\, \phi$ with existentially quantified variables x_{e_1}, ..., x_{e_m} (where $e_1 < \cdots < e_m$) is valid if and only if there is a model $\{x_{e_k} \mapsto f_{e_k}\}_{k=1}^{m}$ such that the propositional formula*

$$\phi[x_{e_m} \mapsto f_{e_m}(x_1, \ldots, x_{e_m - 1})] \cdots [x_{e_1} \mapsto f_{e_1}(x_1, \ldots, x_{e_1 - 1})]$$

obtained by replacing existential variables with their witness functions in ϕ is valid.

Thus, every valid QBF has a model that witnesses its validity, and conversely, a model that produces a valid propositional formula proves validity of the original QBF. This is the theoretical foundation for the certificate format that we describe in Section 3.3.

As a simple example, consider (1). A model is given by $f_y(x) = \bot$ and $f_z(x, y) = x$. From this model, we obtain the propositional formula $(x \vee \bot \vee \neg x) \wedge (x \vee \top \vee x) \wedge (\neg x \vee \bot \vee x) \wedge (\neg x \vee \top \vee \neg x) \wedge (\top \vee x)$. This formula is easily seen to be valid: each of its clauses is valid, containing either $\top$ or both x and $\neg x$. Hence (1) is valid by Theorem 1.

3.3 Certificates of Validity

Squolem generates certificates in a unified format that is described in detail in [32]. The format is ASCII-based. Clauses and variables are indexed by positive integers. Negative values stand for negated variables, i.e., integer negation denotes propositional negation. Indices do not necessarily correspond to variable positions in the quantifier prefix.

A certificate of validity encodes a model of a QBF, as defined in Section 3.2. Certificates introduce fresh *extension variables* as abbreviations for witness functions and other (sub-)formulae. For each extension variable, the certificate contains a line that defines the extension as either

- a conjunction of literals (with the empty conjunction denoting $\top$), or
- a formula if x then y else z, where x, y, z are literals.

All variables that occur in the definiens must be extension variables that have been defined previously, or must come from the original QBF. From these two simple building blocks, extension variables can be defined for arbitrary propositional formulae (and hence, for arbitrary witness functions). The certificate's final line contains a list of (x_k, f_k) pairs that establishes the map from existential variables to witness functions, each function denoted by a (possibly negated) extension variable.

For instance, mapping x, y and z to variable indices 1, 2 and 3, respectively, Squolem generates the following certificate for (1):

```
QBCertificate                  // explanatory comments:
E 4 A 2 0                      // $v_4 = v_2$ (0 ends the line)
E 5 A 1 -2 0                   // $v_5 = v_1 \wedge \neg v_2$
E 6 I 4 4 5                    // $v_6 =$ if $v_4$ then $v_4$ else $v_5$
E 7 A 0                        // $v_7 = \top$ (empty conjunction)
CONCLUDE VALID 2 -7 3 6        // $v_2 = \neg v_7$, $v_3 = v_6$
```

There are four extension variables v_4 through v_7, defined as $v_4 = y$, $v_5 = x \wedge \neg y$, $v_6 =$ if v_4 then v_4 else v_5, and $v_7 = \top$. The witness for y is declared to be $\neg v_7$, i.e., $\bot$. The witness for z is declared to be v_6, which simplifies to x given that $v_4 = y = \bot$. The certificate thus encodes the model given in Section 3.2.

We have written a simple recursive-descent parser for this certificate format that returns the encoded information as a value in Standard ML.

3.4 Higher-Order Logic

HOL4 is a popular LCF-style [10,11] theorem prover for polymorphic higher-order logic [9]. It is based on Church's simple type theory [33] extended with Hindley-Milner style polymorphism [34]. Higher-order logic (HOL) contains a type of Booleans, propositional connectives, and quantifiers over arbitrary types. Hence, quantified propositional logic embeds straightforwardly into HOL.

HOL4 implements a natural-deduction calculus. Theorems represent *sequents* $\Gamma \vdash \phi$, where Γ is a finite set of *hypotheses*, and ϕ is the sequent's *conclusion*. Instead of $\emptyset \vdash \phi$, we simply write $\vdash \phi$. Internally, the set of hypotheses is given by a red-black tree (for efficient search, insertion and deletion), with terms treated modulo α-equivalence.

Like other LCF-style provers, HOL4 has a small inference kernel. Theorems are implemented as an abstract data type, and new theorems can be constructed only through a fixed set of functions provided by this data type. These functions directly correspond to the axiom schemata and inference rules of higher-order logic. Figure 1 shows the rules of HOL that we use to validate certificates of QBF validity (our call to MiniSat [35], described in the next section, may use additional primitive rules involving negation).

$$\frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume}_{\phi} \qquad \frac{\mathit{\Gamma} \vdash \phi}{\mathit{\Gamma}\,\theta \vdash \phi\,\theta}\ \textsc{Inst}_{\theta} \qquad \frac{}{\vdash t = t}\ \textsc{Refl}_{t}$$

$$\frac{\mathit{\Gamma} \vdash \psi}{\mathit{\Gamma} \setminus \{\phi\} \vdash \phi \implies \psi}\ \textsc{Disch}_{\phi} \qquad \frac{\mathit{\Gamma} \vdash \phi \implies \psi \qquad \mathit{\Delta} \vdash \phi}{\mathit{\Gamma} \cup \mathit{\Delta} \vdash \psi}\ \text{MP}$$

$$\frac{\mathit{\Gamma} \vdash \phi}{\mathit{\Gamma} \vdash \forall x.\, \phi}\ \textsc{Gen}_{x}\ (x \text{ not free in } \mathit{\Gamma}) \qquad \frac{\mathit{\Gamma} \vdash \phi[t]}{\mathit{\Gamma} \vdash \exists x.\, \phi[x]}\ \textsc{Exists}_{(\exists x.\, \phi[x],t)}$$

Fig. 1. Selected HOL inference rules

The LCF-style architecture greatly reduces the trusted code base. Proof procedures, although they may implement arbitrarily complex algorithms, cannot produce unsound theorems, as long as the implementation of the theorem data type is correct. HOL4 is written in Standard ML [36], a type-safe functional language (with impure features, e.g., references) that has an advanced module system. To benefit from HOL4's LCF-style architecture, we must implement proof reconstruction in this language.

On top of its LCF-style inference kernel, HOL4 offers various automated proof procedures: e.g., a simplifier, which performs term rewriting, and various first-order provers. The performance of these procedures is hard to control, so we mostly avoid them by combining primitive inference rules directly. A major exception is our use of an existing HOL4 integration [15] of the SAT solver MiniSat to prove the propositional conjecture obtained from a QBF and its validity certificate. We now describe our certificate checking method, with this use of MiniSat, in more detail.

4 Checking Validity Certificates in HOL4

4.1 Overview

Given a QBF $\psi = Q_1 x_1 \ldots Q_n x_n.\, \phi$ and a certificate of its validity, our goal is to derive $\vdash \psi$ as a HOL4 theorem.

The certificate provides witnesses for the QBF's existential variables. However, unfolding the definition of witness functions in the QBF's matrix ϕ, as suggested by Theorem 1, could lead to an exponential blowup of the formula. Instead, we observe that we can use these definitions as hypotheses.

More specifically, the certificate gives a definition $\langle t_i \rangle$ for each extension variable v_i, and a witness literal f_{e_k} (where $\underline{f_{e_k}} = v_i$ for some i) for each existential variable x_{e_k}.[1] We convert definitions $\langle t_i \rangle$ to HOL4 terms t_i, and replace existential variables with their witnesses to ensure each t_i contains only universal and extension variables. We then prove the theorem $\{x_{e_1} \Leftrightarrow f_{e_1}, \ldots, x_{e_m} \Leftrightarrow f_{e_m}, v_1 \Leftrightarrow t_1, \ldots, v_p \Leftrightarrow t_p\} \vdash \phi$, where m and p are the number of existential and extension variables, respectively. To prove validity of the QBF from this theorem, we reintroduce quantifiers in order, from Q_n up to Q_1, and prove the hypotheses. We eliminate hypotheses eagerly, while ensuring we do not unfold the definition of any variable that occurs more than once in the sequent.

In the certificate, variables are indexed by positive integers. Our implementation maintains a one-to-one correspondence between variables and indices. For conceptual clarity, we describe the algorithm entirely in terms of variables. The implementation uses indices where possible to achieve better performance.

We maintain two maps keyed on variables. The first map, V, gives the variable's kind—universal, existential, or extension—along with a HOL4 term for its definition, if applicable: f_{e_k} for existential, and t_i for extension variables. The second map, D, maps each variable to a list of variables that it *depends* on. Dependency between variables is characterized as follows.

(D_1) x_k depends on x_{k+1}, for all $1 \leq k < n$;
(D_2) $\underline{f_{e_k}}$ depends on x_{e_k}, for all $1 \leq k \leq m$; and
(D_3) each variable in t_i depends on v_i, for all $1 \leq i \leq p$.

We explain in Section 4.3 how dependencies are used for hypothesis elimination.

Our algorithm for checking validity certificates has four main steps.

1. Construct the formula $\phi' = (x_{e_1} \Leftrightarrow f_{e_1}) \Rightarrow \cdots \Rightarrow (x_{e_m} \Leftrightarrow f_{e_m}) \Rightarrow \phi$ and partially construct the maps V and D, omitting extension variables;
2. Add extension variable definitions, obtaining the formula $\phi'' = (v_1 \Leftrightarrow t_1) \Rightarrow \cdots \Rightarrow (v_p \Leftrightarrow t_p) \Rightarrow \phi'$, and finish constructing the maps V and D;
3. Prove the (purely propositional) theorem $\vdash \phi''$ using the MiniSat integration, then turn its antecedents into hypotheses to obtain $\{x_{e_1} \Leftrightarrow f_{e_1}, \ldots, x_{e_m} \Leftrightarrow f_{e_m}, v_1 \Leftrightarrow t_1, \ldots, v_p \Leftrightarrow t_p\} \vdash \phi$; and finally,
4. Topologically sort the variables according to D, then eliminate hypotheses and reintroduce quantifiers to obtain $\vdash \psi$.

4.2 Preparing the Formula for MiniSat

We first process the quantifier prefix of ψ, stripping off one quantifier at a time until we obtain ϕ. For each quantifier, we add $x_k \mapsto [x_{k+1}]$ to D (or $x_n \mapsto [\,]$ for the innermost variable). For each universal quantifier, we add $x_k \mapsto \forall$ to V. For each existential quantifier, we add $x_{e_k} \mapsto (\exists, f_{e_k})$ to V and $\underline{f_{e_k}} \mapsto [x_{e_k}]$ to D. When all the quantifiers have been stripped, V maps every quantified variable, and D accurately represents the first two dependency conditions. Iterating over V, we add each existential variable's definition, $x_{e_k} \Leftrightarrow f_{e_k}$, as an antecedent to ϕ to complete the algorithm's first main step, obtaining ϕ'.

[1] Squolem may omit witnesses for variables whose value does not affect the QBF's validity. For these variables we use a dummy extension variable with definition $\top$.

Next, we process the certificate's definitions of extension variables. For each definition $(v_i, \langle t_i \rangle)$, we construct a term t_i by creating a HOL4 conjunction or if-then-else term, replacing references to existential variables with their witnesses. For each variable x that occurs in t_i (after replacing existential variables), we add v_i to the list associated with x in D. Thus, when all definitions have been processed, D accurately represents all three dependency conditions. We also add $v_i \mapsto (\mathsf{ext}, t_i)$ to V, and $v_i \Leftrightarrow t_i$ as an antecedent to ϕ', in the end obtaining ϕ''. This completes the second main step.

We now invoke MiniSat to prove ϕ'', which is a purely propositional formula with antecedents defining all existential and extension variables and the original matrix as consequent. MiniSat is an independent SAT solver that has been integrated into HOL4 just as we are now integrating Squolem. In particular, MiniSat logs proofs, and each proof is replayed via HOL4 inferences to produce a theorem that depends only on the trusted kernel [15]. When MiniSat returns a theorem $\vdash \phi''$, we turn all antecedents into hypotheses using Assume and MP.

4.3 Hypothesis Elimination

Given the theorem $\{x_{e_1} \Leftrightarrow f_{e_1}, \ldots, x_{e_m} \Leftrightarrow f_{e_m}, v_1 \Leftrightarrow t_1, \ldots, v_p \Leftrightarrow t_p\} \vdash \phi$ obtained from the previous step, our goal is to introduce quantifiers and eliminate hypotheses to obtain $\vdash \psi$. To introduce a universal (existential) quantifier, we use Gen (Exists, respectively). To eliminate a hypothesis of the form $x \Leftrightarrow t$, we use Inst with a substitution mapping x to t. The hypothesis thus becomes $t \Leftrightarrow t$. We prove $\vdash t \Leftrightarrow t$ with Refl, then use Disch and MP (see Figure 1).

However, care must be taken to introduce quantifiers and eliminate hypotheses in the correct order. The Inst rule instantiates *all* free occurrences of a variable in a sequent. When eliminating a hypothesis, we want the variable on its left-hand side to occur only there, both to avoid changing the conclusion of the theorem, whose matrix should always be ϕ, and to prevent terms in the hypothesis set from growing too large. Therefore, before eliminating $x \Leftrightarrow t$, we ensure both that x is quantified in the conclusion (or is an extension variable), and that x does not appear on the right-hand side of any hypothesis. It is enough to consider right-hand sides, since the left-hand sides are all distinct.

A variable x has been *eliminated* if it has been quantified, if necessary, and the hypothesis with x on the left, if any, has been eliminated. Only existential variables require both treatments; for them, we quantify before eliminating the hypothesis. A variable x depends on another variable y if y must be eliminated before x can be eliminated. The last two dependency conditions defined in Section 4.1 effectively say that the left-hand side of a hypothesis must be eliminated before any variable on its right-hand side, which agrees with our observations about Inst. Dependency condition D_1 simply ensures that we introduce quantifiers in the correct order. To complete the algorithm's final main step, we topologically sort all variables according to their dependencies in D, then eliminate each variable in the order obtained.

The $\textsc{Gen}_x$ rule has a side condition: x must not occur free in the hypotheses. We rely on the fact that if we eliminate hypotheses eagerly, i.e., as soon as their left-hand side is a lone occurrence, then the side condition holds as long as each witness function f_{e_j} represented by the certificate depends only on variables $x_1, \ldots, x_{e_j-1}$. In fact, Squolem may re-order existential variables, i.e., define a witness f_{e_j} in terms of x_{e_k} for some $e_k > e_j$, provided there is no intervening universal quantifier. However, only acyclic dependencies between existential variables are allowed; cycles are detected as failure of the topological sort.[2]

4.4 Example

Consider (1) again, where we have

$$\psi = \forall x\, \exists y\, \exists z.\ (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z).$$

Assume x, y, and z have variable indices 1, 2, and 3, respectively. Squolem provides the certificate of validity given in Section 3.3. Let v_1 through v_4 be extension variables with indices 4 through 7. Witnesses are given as $(y, \neg v_4)$ and (z, v_3). Definitions are given as $(v_1, \texttt{A}\ y\ \texttt{0})$, $(v_2, \texttt{A}\ x\ {-y}\ \texttt{0})$, $(v_3, \texttt{I}\ v_1\ v_1\ v_2)$, and $(v_4, \texttt{A}\ \texttt{0})$. After processing the quantifier prefix, we have

$$\begin{aligned}
&\phi = (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z),\\
&V = \{x \mapsto \forall, y \mapsto (\exists, \neg v_4), z \mapsto (\exists, v_3)\},\\
&D = \{x \mapsto [y], y \mapsto [z], z \mapsto [\,], v_3 \mapsto [z], v_4 \mapsto [y]\},\\
&\phi' = (y \Leftrightarrow \neg v_4) \Rightarrow (z \Leftrightarrow v_3) \Rightarrow \phi.
\end{aligned}$$

We process the definitions of extension variables (as described in Section 4.2) to obtain $t_1 = \neg v_4$, $t_2 = x \wedge v_4$, $t_3 = \textsf{if}\ v_1\ \textsf{then}\ v_1\ \textsf{else}\ v_2$, $t_4 = \top$, and

$$\begin{aligned}
V = \{&x \mapsto \forall, y \mapsto (\exists, \neg v_4), z \mapsto (\exists, v_3),\\
&v_1 \mapsto (\textsf{ext}, t_1), v_2 \mapsto (\textsf{ext}, t_2), v_3 \mapsto (\textsf{ext}, t_3), v_4 \mapsto (\textsf{ext}, t_4)\},\\
D = \{&x \mapsto [y, v_2], y \mapsto [z], z \mapsto [\,], v_1 \mapsto [v_3], v_2 \mapsto [v_3], v_3 \mapsto [z], v_4 \mapsto [y, v_1, v_2]\},\\
\phi'' = {}&(v_1 \Leftrightarrow t_1) \Rightarrow (v_2 \Leftrightarrow t_2) \Rightarrow (v_3 \Leftrightarrow t_3) \Rightarrow (v_4 \Leftrightarrow t_4) \Rightarrow \phi'.
\end{aligned}$$

After passing ϕ'' to MiniSat and stripping all antecedents from the resulting theorem, we have

$$\{y \Leftrightarrow \neg v_4, z \Leftrightarrow v_3, v_1 \Leftrightarrow t_1, v_2 \Leftrightarrow t_2, v_3 \Leftrightarrow t_3, v_4 \Leftrightarrow t_4\} \vdash \phi$$

We now eliminate variables in a topological order according to D, say $z < v_3 < v_1 < v_2 < y < x < v_4$. After eliminating z, v_3, and v_1, we have

$$\{y \Leftrightarrow \neg v_4, v_2 \Leftrightarrow t_2, v_4 \Leftrightarrow t_4\} \vdash \exists z.\, \phi$$

[2] Squolem's own certificate validator, QBV, did not implement this acyclicity check correctly in version 1.03. It would therefore accept certain invalid certificates. This bug has been fixed in the latest version (2.0) of QBV.

We show the remainder in more detail. To eliminate v_2, we instantiate v_2 to t_2, then prove the hypothesis with a theorem $\vdash t_2 \Leftrightarrow t_2$:

$$\begin{aligned} \{y \Leftrightarrow \neg v_4, t_2 \Leftrightarrow t_2, v_4 \Leftrightarrow t_4\} &\vdash \exists z.\, \phi \\ \{y \Leftrightarrow \neg v_4, v_4 \Leftrightarrow t_4\} &\vdash \exists z.\, \phi \end{aligned}$$

To eliminate y, we first quantify. Then we can instantiate without affecting the sequent's conclusion, before proving the hypothesis:

$$\begin{aligned} \{y \Leftrightarrow \neg v_4, v_4 \Leftrightarrow t_4\} &\vdash \exists y.\, \exists z.\, \phi \\ \{\neg v_4 \Leftrightarrow \neg v_4, v_4 \Leftrightarrow t_4\} &\vdash \exists y.\, \exists z.\, \phi \\ \{v_4 \Leftrightarrow t_4\} &\vdash \exists y.\, \exists z.\, \phi \end{aligned}$$

To eliminate x, we simply quantify, which is possible since all hypotheses mentioning x have been eliminated. And to eliminate v_4, we instantiate again before proving the hypothesis as before.

$$\begin{aligned} \{v_4 \Leftrightarrow t_4\} &\vdash \forall x.\, \exists y.\, \exists z.\, \phi \\ \{t_4 \Leftrightarrow t_4\} &\vdash \forall x.\, \exists y.\, \exists z.\, \phi \\ \emptyset &\vdash \forall x.\, \exists y.\, \exists z.\, \phi \end{aligned}$$

This sequent is now $\vdash \psi$ as required.

5 Experimental Results

We have evaluated our implementation on a set of 100 valid QBF problems that resulted from applying Squolem 2.02 to all 445 problems in the *2005 fixed instance* and *2006 preliminary QBF-Eval* data sets. With a time limit of 600 seconds per problem, Squolem solved 217 of these problems; 100 were determined to be valid.[3] (We did not consider inverting invalid problems.) The same set of problems was previously used (by the Squolem authors) to evaluate the performance of Squolem's certificate generation [7].

All experiments were conducted on a 64-bit Linux system with an Intel Core i7-920XM processor at 2.0 GHz clock speed. Memory usage was restricted to 4 GB. HOL4 was running on top of Poly/ML 5.4.1.

5.1 Run-Times

Table 1 shows our experimental results for the first 50 of the 100 valid QBF problems. Our full results are available at `http://www.cl.cam.ac.uk/~tw333/qbf/`. The remainder of this section comprehensively covers all 100 problems.

The first column in Table 1 gives the name of the benchmark. The next three columns provide information about the size of the benchmark, giving the

[3] In comparison, Squolem 1.03 only solved 142 problems, among them 73 valid ones.

number of alternating quantifiers,[4] variables, and clauses, respectively. Column five shows the run-time of Squolem 2.02 (with certificate generation enabled) to solve the benchmark. Column six shows the number of extension variables in the resulting certificate.

The last two columns finally show the run-time of certificate validation in HOL4. The HOL4 system comes with two different implementations of its inference kernel: one uses de Bruijn indices (and explicit substitutions) to represent λ-terms [37], the other (by M. Norrish) uses a name-carrying implementation [38]. These implementations differ in the performance (and even complexity) of primitive operations. We present run-times for both implementations.

All run-times are given in seconds (rounded to the nearest tenth of a second). Timeouts are indicated by T. For comparison, we have also measured run-times of QBV [7], a stand-alone checker for Squolem's certificates that was developed by the authors of Squolem. QBV is written in C++ and uses MiniSat for tautology checking. Its run-times are given in column seven.

We observe that even for Squolem's stand-alone checker QBV, certificate validation is considerably harder than certificate generation on selected problems (e.g., adder-6-sat, qshifter_7, qshifter_8). This is in line with earlier results [7] and reflects the fact that certificate validation for valid QBF instances is, in general, co-NP-complete [39].

However, QBV times out on one problem only, while HOL4 times out on 13 problems (de Bruijn kernel) or 15 problems (name-carrying kernel). This corresponds to success rates of 87% and 85%, respectively. These rates are largely due to a number of relatively easy problems. The largest certificates that are validated successfully in HOL4 (k_d4_n-20, toilet_c_08_10.2) define just over 15000 extension variables each; QBV validates them in about a second. Figure 2 shows run-times for the de Bruijn kernel as a function of the number of extension variables. The dotted trend line ($R^2 = 0.96$) is given by $f(x) = 1.09 \cdot 10^{-5} \cdot x^{1.85}$.

If we count each timeout as 600 seconds, average run-times are 7.5 seconds for Squolem, 8.4 seconds for QBV, 134.1 seconds for the de Bruijn kernel, and 163.1 seconds for the name-carrying kernel. (Considering successfully validated problems only, average run-times are 2.4 seconds for QBV, 64.5 seconds for the de Bruijn kernel, and 86.1 seconds for the name-carrying kernel.) The de Bruijn kernel thus takes 16 times longer on average than QBV (and 18 times longer than Squolem's proof search), but is almost 18% faster than the name-carrying kernel.

Interestingly, the picture is very different for invalid QBF [24]. For LCF-style validation of invalidity certificates, the name-carrying kernel is 75 times faster on average than the de Bruijn kernel, and 25 times faster than proof search with Squolem. This shows that LCF-style validation of QBF invalidity is not only easier in general, but also exercises different primitive inference rules of the HOL4 kernel.

[4] Counting successive quantifiers of the same kind, as in $\forall x\, \forall y\, \forall z \ldots$, as one quantifier only. The total number of quantifiers in each benchmark is typically identical to the number of variables.

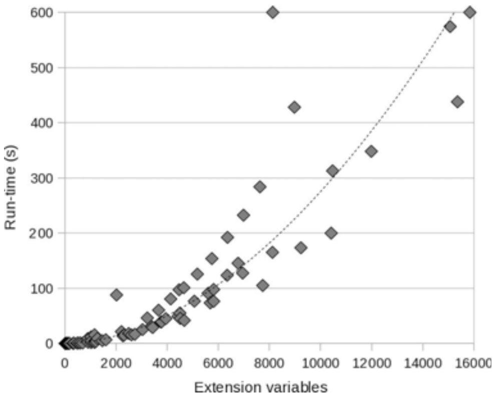


Fig. 2. Run-times/extension variables

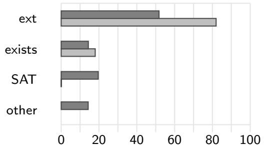


Fig. 3. de Bruijn

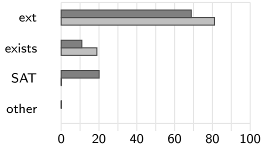


Fig. 4. Name-carrying

5.2 Profiling and Future Improvements

To gain deeper insight into these results, we present profiling data for the de Bruijn kernel (Figure 3) and the name-carrying kernel (Figure 4).

For each kernel, we show the share of total run-time (dark bars) and relative number of function calls (light bars) for the following functions: elimination of extension variable definitions (ext), introduction of existential quantifiers into the conclusion (exists), and propositional tautology proving (SAT). Time spent on other aspects of certificate validation, e.g., parsing and pre-processing the certificate, is shown as well (other). The relative number of function calls (light bars) is similar for each kernel; small differences are caused by timeouts.

We observe that the bulk of run-time (52% for the de Bruijn kernel, 69% for the name-carrying kernel) is spent on eliminating extension variables. There are about four extension variables on average for every existential quantifier. Introducing the latter takes 11% (name-carrying kernel) to 14% (de Bruijn kernel) of total run-time. With either kernel, about 20% of run-time is spent on propositional tautology proving, i.e., on the call to MiniSat with pre-processing and proof checking in HOL4. Other inferences, e.g., introduction of universal quantifiers via Gen, take a much greater share of run-time in the de Bruijn kernel (14%) than in the name-carrying kernel (0%). As indicated by the run-times in Section 5.1, however, this effect is more than outweighed by the name-carrying kernel's slightly inferior performance on ext.

At present, we convert the negation of the QBF's matrix ϕ into CNF. This is costly, despite the fact that HOL4 uses a Tseitin-style transformation [15]. It would be more apt to call MiniSat several times, to prove each clause of ϕ separately from the certificate's definitions. However, the overhead associated with calling MiniSat from HOL4 currently renders this approach infeasible: no incremental interface to MiniSat is available in HOL4.

More substantial improvements might be gained from a modified term data structure. The kernel could compute the set of a term's free variables when the

Table 1. Experimental results

				Squolem	Ext.	QBV	de Bruijn	name-
Benchmark name	Quant.	Vars.	Clauses	(s)	vars.	(s)	(s)	carry. (s)
Adder2-2-s	6	236	292	0.0	352	0.0	0.7	1.0
Adder2-4-s	6	1117	1405	1.3	5685	0.4	73.6	88.3
adder-2-sat	4	51	109	0.0	179	0.0	0.2	0.3
adder-4-sat	4	226	530	0.1	4422	0.2	49.1	56.8
adder-6-sat	4	525	1259	7.8	104897	83.2	T	T
CHAIN12v.13	3	925	4582	0.1	896	0.0	9.0	13.8
CHAIN13v.14	3	1080	5458	0.1	1023	0.0	12.1	19.2
CHAIN14v.15	3	1247	6424	0.1	1157	0.0	15.7	25.2
CHAIN15v.16	3	1426	7483	0.1	3221	0.1	46.3	86.5
CHAIN16v.17	3	1617	8638	0.2	3664	0.1	60.4	113.3
CHAIN17v.18	3	1820	9892	0.3	4136	0.1	80.7	145.4
CHAIN18v.19	3	2035	11248	0.3	4649	0.1	101.1	192.6
CHAIN19v.20	3	2262	12709	0.4	5178	0.1	125.7	247.2
CHAIN20v.21	3	2501	14278	0.5	5748	0.1	153.9	311.4
CHAIN21v.22	3	2752	15958	0.5	6358	0.1	192.3	381.0
CHAIN22v.23	3	3015	17752	0.7	6985	0.1	232.5	467.4
CHAIN23v.24	3	3290	19663	0.9	7628	0.1	283.9	567.8
comp.blif_0.10_0.20_0_1_inp_exact	7	311	833	1.3	9231	0.5	173.5	169.4
comp.blif_0.10_1.00_0_1_inp_exact	3	307	844	0.1	4667	0.2	41.8	46.8
counter_2	5	42	103	0.0	322	0.0	0.2	0.3
counter_4	9	130	333	0.4	7737	0.2	105.1	104.6
counter_e_2	5	50	123	0.0	692	0.0	0.8	1.0
counter_e_4	9	144	373	136.4	69771	7.3	T	T
counter_r_2	5	50	121	0.0	360	0.0	0.3	0.4
counter_r_4	9	144	369	0.8	10415	0.4	200.0	226.0
counter_re_2	5	58	141	0.0	583	0.0	0.6	0.9
counter_re_4	9	158	409	38.8	40716	3.4	T	T
impl02	5	10	18	0.0	14	0.0	0.0	0.0
impl04	9	18	34	0.0	28	0.0	0.0	0.0
impl06	13	26	50	0.0	42	0.0	0.0	0.0
impl08	17	34	66	0.0	56	0.0	0.0	0.0
impl10	21	42	82	0.0	70	0.0	0.0	0.1
impl12	25	50	98	0.0	84	0.0	0.0	0.1
impl14	29	58	114	0.0	98	0.0	0.1	0.1
impl16	33	66	130	0.0	112	0.0	0.1	0.1
impl18	37	74	146	0.0	126	0.0	0.1	0.1
impl20	41	82	162	0.0	140	0.0	0.1	0.2
k_branch_n-4	13	803	2565	33.7	8118	0.5	165.2	235.3
k_d4_n-16	41	1437	5140	0.9	11985	0.8	348.2	561.2
k_d4_n-20	49	1785	6416	1.3	15069	1.1	574.4	T
k_d4_n-21	51	1872	6735	1.4	15840	1.1	T	T
k_d4_n-4	17	393	1312	0.1	2733	0.1	16.6	28.3
k_d4_n-8	25	741	2588	0.3	5817	0.2	76.5	126.9
k_dum_n-12	35	620	1594	0.1	2315	0.1	15.1	25.0
k_dum_n-16	43	796	2062	0.1	3035	0.2	25.2	46.2
k_dum_n-20	51	972	2530	0.1	3755	0.2	38.7	70.0
k_dum_n-21	53	1016	2647	0.2	3945	0.2	45.1	79.1
k_dum_n-4	19	262	649	0.0	902	0.0	2.2	3.8
k_dum_n-8	27	444	1126	0.0	1599	0.0	6.7	12.6
k_grz_n-12	17	557	2003	7.5	4510	0.2	45.3	77.9

term is built, and store it in memory along with the term itself. This would permit more efficient implementations of instantiation and generalization (see Figure 1). However, it is difficult to predict the effect that such a major change in fundamental kernel data structures would have on other HOL4 applications.

As Figure 2 shows, eliminating extension variables is essentially quadratic. Harrison [40] presents an ingenious solution to this kind of problem using proforma theorems to reduce the complexity. Adapting his approach would require some effort, but could yield significant performance improvements.

6 Conclusions

We have presented LCF-style checking for certificates of QBF validity in HOL4. Detailed performance data shows that LCF-style certificate checking is often feasible even for large valid QBF instances: up to 87% of our benchmark certificates were checked successfully. With a time limit of 600 seconds, the algorithm succeeds on certificates that have at most some 15000 extension variables. Our implementation is freely available from the HOL4 repository [38].

Our work complements earlier work on LCF-style checking for certificates of QBF invalidity [24]. It has two main applications. First, it enables HOL4 users to benefit from Squolem's automation. QBF can simply be passed from the HOL4 system to Squolem. If Squolem proves that the QBF is valid, our method then derives it as a theorem in HOL4. Second, our work provides high correctness assurances for Squolem's results; in fact, we uncovered a soundness bug in an earlier version of Squolem's certificate validator QBV. Due to HOL4's LCF-style architecture, our proof checker cannot draw unsound inferences (provided HOL4's kernel is correct). Thus, it can be used for QBF benchmark certification.

One could extend this work to other QBF solvers (see [23] for an overview), and to other interactive theorem provers, e.g., Isabelle or Coq. Because seemingly minor differences in kernel data structures can have significant effects, it is not clear whether similar performance can be achieved in these systems.

An alternative approach that might yield better performance than the LCF-style implementation presented in this paper is the use of reflection [41], i.e., implementing and proving correct a checker for Squolem's certificates in the prover's logic, and then executing the verified checker without producing proofs. While this approach still provides relatively high correctness assurances, obtaining a theorem in HOL4 would require enhancing the inference kernel with a reflection rule that allows us to trust the result of such a verified computation.

Acknowledgments. The authors would like to thank Christoph Wintersteiger for assistance with Squolem and QBV.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

2. Gopalakrishnan, G.C., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 401–413. Springer, Heidelberg (2004)
3. Dershowitz, N., Hanna, Z., Katz, J.: Bounded model checking with QBF. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 408–414. Springer, Heidelberg (2005)
4. Benedetti, M., Mangassarian, H.: QBF-based formal verification: Experience and perspectives. JSAT 5(1-4), 133–191 (2008)
5. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: Proc. 5th Annual ACM Symp. on Theory of Computing, pp. 1–9 (1973)
6. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and certifying QBFs: A comparison of state-of-the-art tools. AI Communications 22(4), 191–210 (2009)
7. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 201–214. Springer, Heidelberg (2007)
8. Slind, K., Norrish, M.: A brief overview of HOL4. [42], 28–32
9. Gordon, M.J.C., Pitts, A.M.: The HOL logic and system. In: Towards Verified Systems. Real-Time Safety Critical Systems Series, vol. 2, pp. 49–70. Elsevier, Amsterdam (1994)
10. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78. Springer, Heidelberg (1979)
11. Gordon, M.: From LCF to HOL: a short history. In: Proof, language, and interaction: essays in honour of Robin Milner, pp. 169–185. MIT Press, Cambridge (2000)
12. Bertot, Y.: A short presentation of Coq. [42], 12–16
13. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. [42], 33–38
14. Owre, S., Shankar, N.: A brief overview of PVS. [42], 22–27
15. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2009)
16. Ge, Y., Barrett, C.: Proof translation and SMT-LIB benchmark certification: A preliminary report. In: 6th International Workshop on Satisfiability Modulo Theories (SMT 2008) (2008)
17. 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)
18. Kumar, R., Kropf, T., Schneider, K.: Integrating a first-order automatic prover in the HOL environment. In: Archer, M., Joyce, J.J., Levitt, K.N., Windley, P.J. (eds.) Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, pp. 170–176. IEEE Computer Society, Los Alamitos (1992)
19. Hurd, J.: An LCF-style interface between HOL and first-order logic. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 134–138. Springer, Heidelberg (2002)
20. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)
21. Letz, R.: Lemma and model caching in decision procedures for quantified boolean formulas. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 5–15. Springer, Heidelberg (2002)
22. Pulina, L., Tacchella, A.: Learning to integrate deduction and search in reasoning about quantified boolean formulas. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 350–365. Springer, Heidelberg (2009)

23. Narizzano, M., Pulina, L., Tacchella, A.: Report of the third QBF solvers evaluation. JSAT 2(1-4), 145–164 (2006)
24. Weber, T.: Validating QBF invalidity in HOL4. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 466–480. Springer, Heidelberg (2010)
25. Büning, H.K., Karpinski, M., Flögel, A.: Resolution for quantified boolean formulas. Information and Computation 117(1), 12–18 (1995)
26. Amjad, H.: Combining model checking and theorem proving. Technical Report UCAM-CL-TR-601, University of Cambridge Computer Laboratory (2004)
27. Ballarin, C.: Computer algebra and theorem proving. Technical Report UCAM-CL-TR-473, University of Cambridge Computer Laboratory (1999)
28. Boldo, S., Filliâtre, J.-C., Melquiond, G.: Combining Coq and Gappa for Certifying Floating-Point Programs. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 59–74. Springer, Heidelberg (2009)
29. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. J. Autom. Reasoning 44(3), 175–205 (2010)
30. QDIMACS standard version 1.1 (2005), `http://www.qbflib.org/qdimacs.html` (released on December 21, 2005) (retrieved February 20, 2011)
31. DIMACS satisfiability suggested format (1993), `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc` (retrieved February 20, 2011)
32. Kroening, D., Wintersteiger, C.M.: A file format for QBF certificates (2007), `http://www.cprover.org/qbv/download/qbcformat.pdf` (retrieved February 20, 2011)
33. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5, 56–68 (1940)
34. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL, pp. 207–212 (1982)
35. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
36. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML – Revised. MIT Press, Cambridge (1997)
37. Barras, B.: Programming and computing in HOL. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 17–37. Springer, Heidelberg (2000)
38. HOL4 contributors: HOL4 Kananaskis 6 source code (2011), `http://hol.sourceforge.net/` (retrieved February 20, 2011)
39. Kleine Büning, H., Zhao, X.: On Models for Quantified Boolean Formulas. In: Lenski, W. (ed.) Logic versus Approximation. LNCS, vol. 3075, pp. 18–32. Springer, Heidelberg (2004)
40. Harrison, J.: Binary decision diagrams as a HOL derived rule. The Computer Journal 38, 162–170 (1995)
41. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge (1995), `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz` (retrieved February 20, 2011)
42. Mohamed, O.A., Muñoz, C., Tahar, S. (eds.): TPHOLs 2008. LNCS, vol. 5170. Springer, Heidelberg (2008)

Proving Valid Quantified Boolean Formulas in HOL Light

Ondřej Kunčar

Charles University in Prague
Faculty of Mathematics and Physics
Automated Reasoning Group
ondrej.kuncar@mff.cuni.cz

Abstract. This paper describes the integration of Squolem, Quantified Boolean Formulas (QBF) solver, with the interactive theorem prover HOL Light. Squolem generates certificates of validity which are based on witness functions. The certificates are checked in HOL Light by constructing proofs based on these certificates. The presented approach allows HOL Light users to prove larger valid QBF problems than before and provides correctness checking of Squolem's outputs based on the LCF approach. An error in Squolem was discovered thanks to the integration. Experiments show that the feasibility of the integration is very sensitive to implementation of HOL Light and used inferences. This resulted in improvements in HOL Light's inference system.

1 Introduction

Deciding whether Quantifier Boolean Formula (QBF) evaluates to true is the canonical PSPACE-complete problem [20]. This problem can be seen as a generalization of the well-known Boolean satisfiability problem (SAT). QBF can contain universal and existential quantifiers over Boolean variables. Let us introduce a simple example, which is nothing else than a definition of the XOR function:

$$\forall v_1\, \forall v_2\, \exists v_3.\, v_3 \Leftrightarrow ((v_1 \wedge \neg v_2) \vee (\neg v_1 \wedge v_2))\ . \qquad (1)$$

Whether the problem of true QBFs is harder than SAT is an open problem. Many problems can be succinctly formulated in QBF – every finite two-player game, many types of planning [7,22], model checking for finite systems and other formal verification problems [2,3,6].

Because we work only with closed formulas, validity and invalidity is the same concept as satisfiability and unsatisfiability respectively. QBF solvers are nowadays powerful tools, which are able to decide validity or invalidity of QBFs automatically. Some of them can generate a certificate that witnesses their output. Squolem [17] is a state-of-the-art QBF solver which is able to generate certificates for valid formulas. These certificates are based on witness functions for existential quantifiers.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 184–199, 2011.

In this paper we present how to prove QBF validity in the HOL Light interactive theorem prover [11,12] using Squolem's certificates of validity. HOL Light, made by John Harrison, is a contemporary interactive theorem prover belonging to the broader family of higher-order logic theorem provers. HOL Light has a very small LCF-style kernel [8] and, moreover, a simplified version of the kernel was proved to be correct [10].

The motivation for our work is twofold. First, interactive theorem provers are nowadays becoming increasingly important thanks to their wide use in areas such as formal specification and verification of complex systems or formalization and verification of mathematics. While these systems often contain a very powerful formalism, their main weakness is that the construction of the proof is often lengthy and requires a considerable human effort. As described in Section 2, many integrations of external tools have been done to increase the amount of automation of interactive theorem provers and to decrease the need for human resources. Each of these integrations resulted in increased strength of the interactive theorem prover – we are talking about situations where formulas that were infeasible to prove using the built-in tactics are proved within a few seconds.

Second, our construction of a proof in HOL Light can serve as another independent check of correctness of Squolem. QBF solvers are generally complex tools with nontrivial implementation in some fast imperative programming language (for example C). This fact causes natural concern about correctness of Squolem. Moreover, it is quite common that QBF solvers disagree on the same inputs. Because HOL Light has a LCF-style kernel, validation of Squolem's certificate in HOL Light lowers significantly the probability that the Squolem's answer was incorrect. We really found a small bug in Squolem due to our system. If a input of Squolem contains tautological clauses, then Squolem 1.0 gives an incorrect answer (and of course an incorrect certificate). Squolem 2.0 gives a correct answer, but still an incorrect certificate. This bug was resolved in the version 2.01 after we pointed out the problem to Christoph Wintersteiger.

Related work is discussed in the next section. In Section 3 we provide necessary definitions and background. We present the main part of our work, how to construct a proof of a valid QBF in HOL Light from Squolem's certificate of validity, in Section 4. We provide experimental results and technical aspects concerning the implementation including optimizations in Section 5. Section 6 concludes this paper and suggests directions for future work.

2 Related Work

The most related work is the paper by Weber [24]. In that paper the author implemented validation of Squolem's certificates of invalidity in another LCF interactive prover HOL4, i.e., it is possible to prove that the given QBF is not valid in HOL4. Squolem's certificates of invalidity are based on a Q-resolution proof of $\perp$. By replaying the resolution proof, a proof of invalidity in HOL4 is established. In principle, it would be possible to prove validity of QBF by using the system by Weber. The method is simple: negate the original formula (valid) and then prove that the negated formula is invalid. But Jussila et al. [17] demonstrated that QBF solvers often perform significantly worse on negated problems. Thus we are going to use directly Squolem's certificates of validity. In the conclusion section of [24] there is a note that LCF-style checking for certificates of validity remains future work. To our knowledge, our work is the first work concerning this task, i.e., proving *valid* QBFs in an interactive theorem prover using an external QBF solver.

Other related work comes from the research area of automation of interactive theorem provers. One of the first integration of an external tool in a trusted theorem prover was the work by Harrison and Thery [13]. It is important to mention earlier but the essential result of John Harrison, who tried to integrate binary decision diagrams (BDD) directly into the HOL Light system [9]. Harrison found out that performing the BDD operations directly by the LCF kernel is about 100 times slower (after optimization) rather than a direct implementation in C. This observation was probably the main reason why most of the further integrations of decision-making procedures use an external solver (to solve the task), which generates a certificate/witness of its output, and a respective proof is generated from such a certificate in the interactive theorem prover.

Let us name just a few recent papers concerning integration of external tools into interactive theorem provers to increase their automation. For first-order theorem provers it is work done by Hurd [15,16] and the Sledgehammer system [19,23]. Weber and Amjad [25] integrated SAT solvers with HOL4, Böhme and Weber integrated the SMT solver Z3 with Isabelle/HOL [4].

Certificates for Squolem were described by Jussila et al. [17]. Other certificates formats were proposed too, an overview can be found in [21]. Authors of Squolem developed a stand-alone checker QBV for Squolem's certificates [17]. It is not surprising that QBV is much more efficient than our approach. On the other hand, QBV would have to become part of the trusted code if users of HOL Light wanted to use it to prove QBFs in HOL Light. Moreover, our system provides a check with much higher assurance than QBV thanks to the LCF-kernel.

3 Theory

3.1 Quantified Boolean Formulas

As usual, we assume that we have an infinite set of Boolean variables. The set of *literals* consists of all variables and their negations. We also extend the notion of negation to literals and identify $\neg\neg v$ with v. A *clause* is a disjunction of literals. A propositional formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. We say that QBF is in *prenex normal form* if the formula is concatenation of a quantifier part and a quantifier-free part. Without loss of generality, we consider only closed QBF in prenex normal form with a propositional core in CNF – the formal definition is as follows:

Definition 1 (Quantified Boolean Formula). *A* Quantified Boolean Formula (QBF) *is a formula of the following form*

$$Q_1 x_1 \dots Q_n x_n . \, \phi \ ,$$

where $n \geq 0$, each x_i is a Boolean variable, each Q_i is the universal $\forall$ or the existential $\exists$ quantifier, and ϕ is a propositional formula in CNF and all of its variables are among $x_1, \ldots, x_n$.

$Q_1 x_1 \ldots Q_n x_n$ is called a *quantifier prefix* and ϕ is called a *matrix*. We define an order $<$ over variables such that $x_1 < x_2$ if x_2 is in the scope of x_1. We call a variable the *intermost* or the *outermost* variable if it is maximal or minimal among all variables of formula (with respect to the order $<$) respectively. We say that x has a *quantification level i* if it is on the i-th position in the quantifier prefix.

If we consider a quantifier prefix as a finite sequence of quantifiers, we can define two relations on quantifier prefixes $\subseteq$ and $\preceq$. We define $\mathbf{Q}_1 \subseteq \mathbf{Q}_2$ if the quantifier prefix $\mathbf{Q}_1$ is a subsequence of the quantifier prefix $\mathbf{Q}_2$, and $\mathbf{Q}_1 \preceq \mathbf{Q}_2$ if the sequence of the variables in $\mathbf{Q}_1$ is a subsequence of the sequence of the variables in $\mathbf{Q}_2$. In other words, $\mathbf{Q}_1 \preceq \mathbf{Q}_2$ if we omit symbols for quantifiers (only variables left) in $\mathbf{Q}_1$ and $\mathbf{Q}_2$, and then we ask if the former is a subsequence of the latter.

The semantics $[\![f]\!]$ of closed QBF f is defined recursively by expanding the outermost variable x: $[\![\forall x.\, \phi]\!] = [\![\phi[x \mapsto 1] \wedge \phi[x \mapsto 0]]\!]$, and similarly $[\![\exists x.\, \phi]\!] = [\![\phi[x \mapsto 1] \vee \phi[x \mapsto 0]]\!]$. Where $\phi[x \mapsto c]$ denotes ϕ in which every free occurrence of x is replaced by the constant c. We call QBF *valid* or *invalid* if its semantics is 1 or 0 respectively.

3.2 QBF Models

Squolem's certificate of validity contains a model of the given QBF. The following general definition of a QBF model is a slightly improved definition used in [5,17].

Definition 2 (Model). *Let $\Phi = Q_1 x_1 \ldots Q_n x_n.\, \phi$ be a valid closed QBF in prenex normal form. Let V_i be the set of variables of Φ that have their quantification level less than or equal to i and let E_i and A_i be the sets of the existentially and universally quantified variables in V_i respectively, i.e., $E_i \cup A_i = V_i$. Let M be the set of functions*

$$M := \{f_{v_k} : \{0,1\}^{k-1} \to \{0,1\} \mid v_k \in E_n\},$$

where each f_{v_k} depends exactly on the $k-1$ variables from V_{k-1}. M is said to be a model of Φ if

$$[\![\forall x_{i_1} \ldots \forall x_{i_k}.\, \phi\, [x_{j_1} \mapsto f_{x_{j_1}}(x_1, \ldots, x_{j_1 - 1}), \ldots, x_{j_l} \mapsto f_{x_{j_l}}(x_1, \ldots, x_{j_l - 1})]]\!] = 1,$$

where $\{x_{i_1}, \ldots, x_{i_k}\} = A_n$ and $\{x_{j_1}, \ldots, x_{j_l}\} = E_n$.

Functions f_v in the definition are nothing else than *witness functions*, which give witnesses based on (potentially) all preceding variables. As is noted in [17], it is also possible to let functions f_{v_k} only depend on the universally quantified variables of V_{k-1} but the authors of [17] claim that the stated definition may result in more compact representations of the functions f_{v_k}. In practice these functions are represented by propositional formulas.

3.3 Squolem's Certificates of Validity

Squolem's certificate format is described in detail in [18], we describe only the relevant part – certificates for valid formulas. The format is text based, variables are represented by positive integers and negated variables are denoted by negative integers, i.e., integer negation expresses propositional negation. The certificate describes a model of a given valid QBF by providing witness functions for existentially quantified variables.

The functions are defined gradually by *extensions*: definitions that introduce new Boolean functions defined by propositional formulas. Each new extension introduces a fresh variable which can be later used for referring to the newly defined Boolean function. It is a reasonable requirement not to allow an arbitrary propositional formula in the definition (which would be too hard to verify), therefore in [18] the authors allow just two special types:

If-Then-Else A new function

$$f(x, y, z) = \mathit{if}\ l_1\ \mathit{then}\ l_2\ \mathit{else}\ l_3$$

is defined as If-Then-Else of three existing variables, where l_1, l_2 and l_3 are literals in variables x, y and z. This function is not actually denoted in the certificate by $f(x, y, z)$, but by a newly introduced fresh variable, let us say w. Then this type of Boolean functions can be represented by the following propositional formula: $w \Leftrightarrow (l_1 \wedge l_2) \vee (\neg l_1 \wedge l_3)$.

And A new function

$$f(x_1, \ldots, x_n) = \bigwedge_{i=1}^{n} l_i$$

is defined as a conjunction of the n literals l_i, which use the variables x_i. The number of conjuncts, n, can be an arbitrary non-negative integer. In the case when $n = 0$ the defined function is the Boolean constant 1. The newly defined function is also actually denoted by a fresh variable and its representation by a Boolean formula is straightforward in this case.

After definitions of all extensions there is a final line containing a list of pairs (v, l_v) for all existentially quantified variables in the given formula. We call the pairs as *witness assignments*. Here v is the existentially quantified variable and l_v is a literal representing an already defined extension, i.e., a possibly negated variable denoting an extension. The corresponding Boolean formula is obvious $v \Leftrightarrow l_v$. This list of witness assignments represents a model in the sense of Definition 2.

Let us conclude this section by an example of Squolem's certificate of validity for formula (1), which is translated into CNF as follows

$$\forall v_1\, \forall v_2\, \exists v_3.\, (v_3 \vee v_1 \vee \neg v_2) \wedge (v_3 \vee v_2 \vee \neg v_1) \wedge (v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_3 \vee \neg v_1 \vee \neg v_2)\ . \quad (2)$$

Squolem then generates the following certificate:

```
QBCertificate
E 4 A 1 -2 0
E 5 A -1 2 0
E 6 I 4 4 5
CONCLUDE VALID 3 6
```

Lines beginning with `E` represent extensions. These three lines represent three extensions, the first two lines define And extensions and the third line is the If-Then-Else extension. The corresponding Boolean formulas are as follows:

$$\begin{gathered} v_4 \Leftrightarrow v_1 \wedge \neg v_2 \\ v_5 \Leftrightarrow \neg v_1 \wedge v_2 \\ v_6 \Leftrightarrow (v_4 \wedge v_4) \vee (\neg v_4 \wedge v_5) \end{gathered}$$

The last line of the certificate says that the witness function for the existentially quantified variable v_3 is the extension v_6. It is not difficult to see that the extension v_6 together with extensions v_4 and v_5 defines the Boolean function XOR.

4 System Description

The overall structure of our system is as follows: first of all, we preprocess the given formula and serialize it into Squolem's input format. We run Squolem, which generates the corresponding certificate of validity. Then we parse this certificate and finally we construct a proof in HOL Light from information gained during the parsing. In this section we describe preprocessing of the given formula and especially construction of the proof. Other parts of our system contain non-interesting software engineering.

4.1 Preprocessing

As our system supports general closed QBFs and Squolem only works with formulas in CNF and prenex normal form, we had to incorporate a preprocessing phase. We implemented a naïve version of the transformation using conversions already available in HOL Light – `NNFC_CONV`, `CNF_CONV` and `PRENEX_CONV`. The transformation may cause an exponential blowup of the formula. More sophisticated conversions could be implemented as well, but because the main focus of this paper is on proof reconstruction (and our benchmark problems are already in prenex CNF), such techniques are beyond the scope of this paper.

The second preprocessing step that was incorporated is renaming of all variables according to the same scheme. We use the scheme `v_i` where `i` is a number representing quantification level of the variable. The scheme provides a uniform way of mapping variables to integers and vice versa, which is useful for text based communication with Squolem (i.e., serializing input and parsing certificates) and in data structures involving variables.

Thus our preprocessing makes the theorem $\vdash \Phi^* \Leftrightarrow \Phi$ where Φ is the original formula and Φ^* is the preprocessed one, which is in the form $Q_1 v_1 \ldots Q_n v_n.\, \phi$.

Our goal is to prove $\vdash \Phi^*$ as a HOL Light theorem given a Squolem's certificate of its validity. The original formula is then trivially inferred by the `EQ_MP` inference.

4.2 Validating Squolem's Model

The question is how to represent the model contained in the given Squolem certificate. We represent a model as the conjunction of the corresponding Boolean formulas of all extensions and witness assignments. Let us denote this term by $\mathfrak{M}$, and call it a *model term.*

For the certificate of formula (1) the model term is defined as follows

$$\mathfrak{M} = (v_4 \Leftrightarrow v_1 \wedge \neg v_2) \wedge (v_5 \Leftrightarrow \neg v_1 \wedge v_2) \wedge (v_6 \Leftrightarrow (v_4 \wedge v_4) \vee (\neg v_4 \wedge v_5)) \wedge (v_3 \Leftrightarrow v_6) \ .$$

Now we can show how to verify the given model. Let us consider the following formula

$$\mathfrak{M} \Rightarrow \phi \ . \tag{3}$$

We claim that the given model is really a model of Φ^* if and only if (3) is a propositional tautology. It is an easy observation that the value of each variable that represents a witness function is uniquely determined in every satisfying assignments of variables on which the function depends. Let us suppose that (3) is not a propositional tautology then the negation of (3) $\mathfrak{M} \wedge \neg\phi$ has a satisfying assignment. This assignment uniquely determines values of existentially quantified variables in ϕ, but does not satisfy ϕ. Thus we find a counterexample that witnesses that the given model is not actually a model of Φ^*. On the other hand, if (3) is a propositional tautology then every satisfying assignment of $\mathfrak{M}$ has to satisfy ϕ.

We prove (3) by calling an external SAT solver. For this we followed Weber and Amjad [25], who integrated external SAT solvers zChaff and MiniSat with HOL theorem provers including HOL Light. In order to prove a formula to be a propositional tautology they negate it and run a SAT solver. If the formula is really a tautology then there is no satisfying assignment of the negated formula and the SAT solver produces a resolution proof of $\perp$. If we replay the resolution proof in the interactive theorem prover, we get our formula as HOL Light's theorem:

$$\vdash \mathfrak{M} \Rightarrow \phi \ . \tag{4}$$

4.3 Adding Quantifiers

Let us denote the quantifier prefix of the formula Φ^* as $\mathbf{Q}$, thus we have the following equation: $\mathbf{Q} = Q_1 v_1 \ldots Q_n v_n$. As was described in 3.3, each And and If-Then-Else extension defines a new fresh variable. We want to define *extended quantifier prefix* $\mathbf{Q_e}$ that correctly incorporates these new fresh variables into $\mathbf{Q}$. We follow [17] and quantify new variables existentially. An important question is how to order the new variables with respect to the variables in the original quantifier prefix – it is clear that they have to be put after the variables on which their extension function depends. But they can't be put too deep because then the corresponding function could depend on variables for which it should serve as a model. Therefore we put each new variable right after the variable with the highest quantification level in the extension function.

This method, however, still doesn't yield a fully correct quantifier prefix. There is a problem with witness assignments. If we have for example a pair (v_i, v_j), the value of v_i depends on v_j. But v_i is not a new fresh variable, it is an existentially quantified variable from the original quantifier prefix $\mathbf{Q}$. Therefore it generally doesn't have to be after the variable v_j. Let us consider an example where $\mathbf{Q} = \forall v_1\, \exists v_2\, \exists v_3$, and there are the extension $v_4 = v_1 \wedge v_3$ and the witness assignment $v_2 = v_4$. After we incorporate v_4, we have $\mathbf{Q_e} = \forall v_1\, \exists v_2\, \exists v_3\, \exists v_4$. But a value of v_2 depends on a value of v_4, therefore v_2 has to be after v_4. Fortunately, it is logically correct to reorder quantifiers in the block of the same quantifiers, thus we can move v_2 after v_4 in our example. In general, we need to topologically sort each block of existential quantifiers according to their extension dependencies.[1]

Our next step is to prove the formula $\mathbf{Q_e}\mathfrak{M} \Rightarrow \mathbf{Q}\phi$. We prove it from (3) by sequential addition of quantifiers by the following three inferences, which we designed and implemented (see 5.1):

$$\frac{\vdash A \Rightarrow B}{\vdash (\forall x.\, A) \Rightarrow \forall x.\, B} \qquad \frac{\vdash A \Rightarrow B}{\vdash (\exists x.\, A) \Rightarrow B}\ (x \text{ not free in B}) \qquad \frac{\vdash A \Rightarrow B}{\vdash A \Rightarrow \exists x.\, B}$$

We go simultaneously through $\mathbf{Q_e}$ and $\mathbf{Q}$, in a bottom-up fashion, and in each step we use the first rule from the following list that matches:

- $\mathbf{Q} = \ldots \exists v_i$ – The whole existential block in $\mathbf{Q}$ will be sequentially added by the third inference. Because $\mathbf{Q_e}$ was made from $\mathbf{Q}$, there has to be the corresponding block in $\mathbf{Q_e}$. It contains the same variables as the corresponding block in $\mathbf{Q}$ plus potentially some fresh variables from extensions. We add this block of $\mathbf{Q_e}$ sequentially by the second inference. The condition 'x not free in B' is satisfied because all common variables from the added blocks of $\mathbf{Q}$ and $\mathbf{Q_e}$ are bounded in B from the first step of this rule.
- $\mathbf{Q_e} = \ldots \exists v_i$ – There is an existential block in $\mathbf{Q_e}$ that contains only fresh variables from extensions, therefore we can add this block in $\mathbf{Q_e}$ by the second inference. If the block contained a non-fresh variable, the first rule from this list would match.
- $\mathbf{Q_e} = \ldots \forall v_i$ and $\mathbf{Q} = \ldots \forall v_i$ – Let us note that the universally quantified variables in $\mathbf{Q_e}$ and $\mathbf{Q}$ have to be the same because we didn't change order of variables in the universally quantified blocks. Thus we add both quantifiers at once by the first inference.

After this juggling with quantifiers we have the following theorem:

$$\vdash \mathbf{Q_e}\mathfrak{M} \Rightarrow \mathbf{Q}\phi\ . \tag{5}$$

[1] This is possible because Squolem never generates circular dependency between extensions and witness assignments.

4.4 Proving the Quantified Model Term

If we were able to prove $\mathbf{Q}_{\mathbf{e}}\mathfrak{M}$, we would accomplish our goal because we can derive $\vdash \mathbf{Q}\phi$ from (5) by a call of MP. And because $\mathbf{Q}\phi = \varPhi^*$, we would be done.

We start with the proof of each extension and witness assignment Boolean formula. For each such a Boolean formula $v_k = \vartheta(v_{i_1}, \dots, v_{i_l})$ we prove the following theorem

$$\vdash \forall v_{i_1} \dots \forall v_{i_l}\, \exists v_k.\, v_k = \vartheta(v_{i_1}, \dots, v_{i_l}) \tag{6}$$

by the following derivation

$$\dfrac{\dfrac{\dfrac{}{\vdash \vartheta(v_{i_1}, \dots, v_{i_l}) = \vartheta(v_{i_1}, \dots, v_{i_l})}\ \mathrm{REFL}_{\vartheta(v_{i_1}, \dots, v_{i_l})}}{\vdash \exists v_k.\, v_k = \vartheta(v_{i_1}, \dots, v_{i_l}))}\ \mathrm{CHOOSE}_{\exists v_k.\, v_k = \vartheta(v_{i_1}, \dots, v_{i_l}),\ \vartheta(v_{i_1}, \dots, v_{i_l})}}{\vdash \forall v_{i_l}\, \exists v_k.\, v_k = \vartheta(v_{i_1}, \dots, v_{i_l})}\ \mathrm{GEN}_{v_{i_l}}$$

$$\dfrac{\vdots}{\vdash \forall v_{i_1} \dots \forall v_{i_l}\, \exists v_k.\, v_k = \vartheta(v_{i_1}, \dots, v_{i_l})}\ \mathrm{GEN}_{v_{i_1}}.$$

It is very important to note that we ordered the variables $v_{i_1}, \dots, v_{i_l}$ according to their quantification levels in $\mathbf{Q}_{\mathbf{e}}$. Let $E_1, \dots, E_N$ be all extension and witness assignment Boolean functions and let $\mathbf{Q}_1, \dots, \mathbf{Q}_N$ be the quantifier prefixes that we get in expressions (6) for each Boolean function. Because we made $\mathbf{Q}_{\mathbf{e}}$ in 4.3 so that each existential variable goes after the variables on which it depends and because we ordered the variables $v_{i_1}, \dots, v_{i_l}$ in the same order as they appear in $\mathbf{Q}_{\mathbf{e}}$, we get for all i that $\mathbf{Q}_i \preceq \mathbf{Q}_{\mathbf{e}}$.

We can rewrite expressions (6) as $\mathbf{Q}_i E_i$ for each extension. Let us consider the following formula $(\mathbf{Q}_1 E_1) \wedge \dots \wedge (\mathbf{Q}_N E_N)$. We show in the rest of this section that we are able to "lift" each $\mathbf{Q}_i$ in front of the big conjunction of E_i's in such a way that we get the following theorem

$$\vdash \mathbf{Q}_{\mathbf{e}}(E_1 \wedge \dots \wedge E_N)\,, \tag{7}$$

which is nothing else than $\vdash \mathbf{Q}_{\mathbf{e}}\mathfrak{M}$.

We designed and implemented the following inference

$$\dfrac{\mathbf{Q}'A \qquad \mathbf{Q}''B}{\mathbf{Q}'''(A \wedge B)}\ \mathrm{LIFT},$$

which has the following property: if $\mathbf{Q}' \preceq \mathbf{Q}_{\mathbf{e}}$ and $\mathbf{Q}'' \preceq \mathbf{Q}_{\mathbf{e}}$, then $\mathbf{Q}''' \preceq \mathbf{Q}_{\mathbf{e}}$. In addition, these two relations hold unconditionally: $\mathbf{Q}' \preceq \mathbf{Q}'''$ and $\mathbf{Q}'' \preceq \mathbf{Q}'''$. With LIFT it is almost possible (we derive $\mathbf{Q}^*$ instead of $\mathbf{Q}_{\mathbf{e}}$) to derive (7) by $N-1$ calls of LIFT:

$$\dfrac{\vdash \mathbf{Q}_{N-2}E_{N-2} \qquad \dfrac{\vdash \mathbf{Q}_{N-1}E_{N-1} \qquad \vdash \mathbf{Q}_N E_N}{\vdash \mathbf{Q}'(E_{N-1} \wedge E_N)}\ \mathrm{LIFT}}{\vdash \mathbf{Q}''(E_{N-2} \wedge E_{N-1} \wedge E_N)}\ \mathrm{LIFT}$$

$$\dfrac{\vdash \mathbf{Q}_1 E_1 \qquad \dfrac{\ddots \qquad \vdots}{\vdash \mathbf{Q}^{'\cdots'}(E_2 \wedge \dots \wedge E_{N-2} \wedge E_{N-1} \wedge E_N)}\ \mathrm{LIFT}}{\vdash \mathbf{Q}^*(E_1 \wedge E_2 \wedge \dots \wedge E_{N-2} \wedge E_{N-1} \wedge E_N)}\ \mathrm{LIFT}$$

Now we finish a proof of (7). From the above written properties of LIFT it follows that $\mathbf{Q}^* \preceq \mathbf{Q_e}$. Because we have for every existentially quantified variable the corresponding extension[2], and because LIFT prefers existentially quantified variables during lifting (see next section), each $\mathbf{Q}_i$ contributes by exactly one existentially quantified variable into $\mathbf{Q}^*$. Therefore $\mathbf{Q}^*$ contains the same existentially quantified variables as $\mathbf{Q_e}$. From that follows that $\mathbf{Q}^* \subseteq \mathbf{Q_e}$. This generally does not have to be equality because some universally quantified variables from $\mathbf{Q_e}$ can be missing in $\mathbf{Q}^*$. Those are exactly the variables that weren't present in any extension, i.e., they are not free in $\mathfrak{M}$, and therefore they can be quite easily added in $\mathbf{Q}^*$. Our rule ADD_MISSING_UNIVERSALS does this job – it is a simple use of HOL Light's rewriting conversions:

$$\frac{\vdash \mathbf{Q}^*(E_1 \wedge \cdots \wedge E_N)}{\vdash \mathbf{Q_e}(E_1 \wedge \cdots \wedge E_N)}\ \text{ADD\_MISSING\_UNIVERSALS}$$

4.5 LIFT

The main goal of LIFT is to prove the following implication

$$\vdash (\mathbf{Q}'A \wedge \mathbf{Q}''B) \Rightarrow \mathbf{Q}'''(A \wedge B)\,. \tag{8}$$

If we have (8), it is straightforward to derive the conclusion of LIFT by a call of CONJ and MP.

Because $\mathbf{Q}' \preceq \mathbf{Q_e}$ and $\mathbf{Q}'' \preceq \mathbf{Q_e}$, all we need to do is to *merge* $\mathbf{Q}'$ and $\mathbf{Q}''$ together according to quantification levels. If we find items of $\mathbf{Q}'$ and $\mathbf{Q}''$ that have different quantifiers, we prefer existential quantifier. We start with $\vdash A \wedge B \Rightarrow A \wedge B$, which we derive by ASSUME$_{A\wedge B}$ and DISCH_ALL.

Then we go simultaneously through $\mathbf{Q}'$ and $\mathbf{Q}''$, in a bottom-up fashion, and perform merging using the following inferences, which we implemented:

$$\frac{\vdash (A \wedge B) \Rightarrow C}{\vdash ((\forall x.\, A) \wedge \forall x.\, B) \Rightarrow \forall x.\, C}$$

$$\frac{\vdash (A \wedge B) \Rightarrow C}{\vdash ((\exists x.\, A) \wedge \forall x.\, B) \Rightarrow \exists x.\, C} \qquad \frac{\vdash (A \wedge B) \Rightarrow C}{\vdash ((\forall x.\, A) \wedge \exists x.\, B) \Rightarrow \exists x.\, C}$$

$$\frac{\vdash (A \wedge B) \Rightarrow C}{\vdash ((\forall x.\, A) \wedge B) \Rightarrow \forall x.\, C}\ x \notin B \qquad \frac{\vdash (A \wedge B) \Rightarrow C}{\vdash ((\exists x.\, A) \wedge B) \Rightarrow \exists x.\, C}\ x \notin B$$

$$\frac{\vdash (A \wedge B) \Rightarrow C}{\vdash (A \wedge \forall x.\, B) \Rightarrow \forall x.\, C}\ x \notin A \qquad \frac{\vdash (A \wedge B) \Rightarrow C}{\vdash (A \wedge \exists x.\, B) \Rightarrow \exists x.\, C}\ x \notin A$$

The notation '$x \notin B$' means 'x is not free in B'. For example, if we encounter two universal quantifiers with the same variables, we use the first rule. On the

[2] If Squolem doesn't generate a witness function for some existentially quantified variable v_i, we add the following artificial extension $v_i \Leftrightarrow 1$.

other hand, if we need to merge two universal quantifiers with different variables and the first variable has higher quantification level than the second, we use the fourth rule and so on.

It is an important observation that we cannot encounter two existential quantifiers with the same variable. As was discussed in 4.4, each $\mathbf{Q}_i$ contributes by exactly one existentially quantified variable and all these variables are different. If we encountered this two-existentials situation, it would be a problem because it generally doesn't hold that $((\exists x.\, A) \wedge \exists x.\, B) \Rightarrow \exists x.\, A \wedge B$.

5 Implementation and Evaluation

5.1 Implementation of Rules

HOL Light has a very simple kernel especially in comparison with HOL4 or Isabelle/HOL. Many rules are not included in HOL Light's kernel and they are derived from primitive rules, including for example the rule MP – modus ponens. It turns out to be one of the sources of inefficiency. We discussed three rules for adding quantifiers into (3) in 4.3. It is natural to implement the second rule by HOL Light's CHOOSE and the third rule by EXISTS.[3] We tried it but this approach turned out to be significantly slower than the following approach: We prove the following schematic theorems

$$\vdash (\forall x.\, A \Rightarrow B) \Rightarrow ((\forall x.\, A) \Rightarrow \forall x.\, B) \qquad \vdash (A \Rightarrow B) \Rightarrow (A \Rightarrow \exists x.\, B)$$

$$\vdash (\forall x.\, A \Rightarrow B) \Rightarrow (\exists x.\, A \Rightarrow B),\ \ x \text{ not free in } B,$$

and in every call of the corresponding rules from 4.3, we instantiate them properly and by MP derive the consequent. We used similar approach in the implementation of rules used in LIFT.

5.2 Alpha-Equivalence Optimization

After we implemented optimizations described in 5.1, performance was still poor. We did some profiling to gain a deeper insight into this problem, and made a quite surprising discovery. Our system spent 99.4 % of the time in HOL Light's kernel function `alphaorder`. This function implements the order of HOL Light's terms with the property that alpha-equivalent terms are equal according to this order. This order is among others used to implement the simple test that two terms are alpha-equivalent. The test for alpha-equivalence is a common test in HOL Light's rules; for example, it is used in the MP rule.

The implementation of `alphaorder t1 t2` is as follows: go simultaneously through (up to bottom) the structure of `t1` and `t2` and compare recursively smaller parts. A list of pairs of alpha-equivalent bound variables is maintained during the traversal. This traversal is implemented in the function `orda`. If $\lambda x.\, s_1$ and $\lambda y.\, s_2$ are compared, then a new pair of alpha-equivalent variables (x, y) is

[3] Both of rules are for example implemented directly in the HOL4 kernel [14].

Table 1. Profiling results

	non-optimized		optimized	
function	relative time (%)	number of calls	relative time (%)	number of calls
`orda`	99.4	668974	16.63	668974
`ordav`	97.84	19786610	2.59	125146
`compare` and `==`	92.33	1225276114	13.67	951183

added to the front of the list. If we need to compare two variables, we have to check the list of alpha-equivalent variables first, which is done in linear time. The comparison of variables is implemented in the function `ordav`.

This linear-time implementation is ineffective for formulas with many abstractions. Because the test for alpha-equivalence of two variables is linear, the test for the whole formula is quadratic. It seems that this is not a problem in normal use of HOL Light (i.e., if common formulas are used). But we work with formulas which have thousands of variables, and because we work only with closed formulas and each quantifier is encoded by a particular type of abstraction, our formulas have thousands of abstractions.

Our optimization is based on the observation of the problem that alpha-equivalence of two identical terms is still possibly quadratic because the pair (x, y) is added to the list even if x and y are identical variables. Thus our optimization is as follows: we detect if the list of alpha-equivalent variables contains pairs of identical variables. If so, we do not use this list. Thus comparison of two identical formulas is linear and not quadratic because all pairs of variables are only compared, and there is no need to go through the list in linear time. The complexity can be actually improved even more because if we do not take the list of alpha-equivalent variables into consideration, we can compare shared subterms only by comparing two pointers pointing to this shared subterm. And this pointer comparison is a constant time operation.

Thanks to this optimization we get a speed-up factor of 321.0 (see 5.3). Detailed profiling data can be seen in Table 1.

5.3 Run-Times

We performed a set of benchmarks to show performance of our implementation and feasibility of validation of Squolem's certificates in HOL Light. We used a similar methodology as in [17,24]. The authors of Squolem conducted experiments on the *2005 fixed instance* and the *2006 preliminary QBF-Eval* data sets, in total 445 instances of QBFs [17]. We ran Squolem with the time limit of 600 seconds and the memory limit of 1 GB. Squolem solved 100 valid problems within the given limits. We ran our system on these 100 valid QBF problems.

All benchmarks were run on a Linux system with four AMD Phenom II X4 955 processors (3.2 GHz) and with 8 GB RAM. We set time limits to 5, 60, 600 and 3000 seconds and the memory limit to 1.5 GB RAM. We present our results for these time limits in Table 3. One can see in the table that we are able to solve more than half of our instances within the time limit of 60 seconds and the

Table 2. Detailed evaluation results for the time limit of 60 seconds

instance name	qntfs.	vars.	clauses	extensions	non. opt. (s)	SAT (s)	model (s)	total (s)
Adder2-2-s	6	249	292	580	179.7	0.3	0.3	1.3
adder-2-sat	4	64	109	206	16.5	0.3	0.1	0.5
CHAIN12v.13	3	925	4582	1809	∞	3.6	2.0	30.7
CHAIN13v.14	3	1080	5458	2090	∞	4.6	2.6	45.6
comp.blif_0.10_1.00_0_1_inp_exact	3	307	844	4973	∞	4.9	30.6	59.1
counter_2	5	42	103	362	40.3	0.2	0.2	0.5
counter_e_2	5	50	123	740	395.1	0.3	0.5	1.3
counter_r_2	5	50	121	408	56.6	0.2	0.2	0.6
counter_re_2	5	58	141	639	228.6	0.3	0.4	1.1
impl02	5	10	18	22	0.0	0.0	0.0	0.0
impl04	9	18	34	42	0.1	0.0	0.0	0.0
impl06	13	26	50	62	0.4	0.0	0.0	0.1
impl08	17	34	66	82	0.7	0.1	0.0	0.1
impl10	21	42	82	102	1.1	0.1	0.0	0.2
impl12	25	50	98	122	1.9	0.1	0.0	0.2
impl14	29	58	114	142	3.0	0.1	0.0	0.2
impl16	33	66	130	162	4.0	0.1	0.1	0.3
impl18	37	74	146	182	6.6	0.1	0.1	0.4
impl20	41	82	162	202	7.5	0.2	0.1	0.5
k_d4_n-4	17	393	1312	3105	∞	4.9	7.2	26.7
k_dum_n-12	35	620	1594	2911	∞	3.9	5.7	30.1
k_dum_n-16	43	796	2062	3799	∞	9.1	9.5	50.5
k_dum_n-4	19	262	649	1152	1400.7	0.8	1.0	4.6
k_dum_n-8	27	444	1126	2023	∞	1.7	2.7	13.2
k_grz_n-4	17	317	902	1767	∞	1.9	2.3	10.5
k_grz_n-8	17	433	1413	3050	∞	3.9	7.6	29.4
k_path_n-12	29	876	2440	4235	∞	4.3	12.2	58.2
k_path_n-4	13	324	888	1464	2839.2	1.2	1.5	7.1
k_path_n-8	21	600	1664	2846	∞	2.4	5.4	26.6
k_ph_n-4	5	141	411	726	328.7	0.5	0.6	2.1
k_poly_n-4	29	330	743	1513	2956.5	1.4	1.6	7.3
k_poly_n-8	53	654	1475	3097	∞	4.4	6.5	31.1
k_t4p_n-4	27	624	1895	4058	∞	10.3	12.5	55.1
mutex-16-s	2	1378	1779	3523	∞	2.4	7.5	32.1
mutex-2-s	2	104	127	214	10.8	0.1	0.1	0.3
mutex-4-s	2	286	363	612	223.5	0.4	0.4	1.6
mutex-8-s	2	650	835	1652	∞	0.9	1.9	7.3
qshifter_3	2	19	128	128	4.2	0.2	0.1	0.3
qshifter_4	2	36	512	512	194.0	1.3	0.4	2.6
qshifter_5	2	69	2048	2048	∞	8.8	4.1	30.8
s27_d2_s	3	65	142	166	4.7	0.1	0.1	0.2
s298_d2_s	3	699	1895	1469	2526.8	1.5	1.2	12.6
s499_d2_s	3	950	2665	2093	∞	2.9	3.5	27.0
TOILET2.1.iv.4	3	37	99	89	0.8	0.1	0.0	0.1
tree-exa10-10	2	20	18	19	0.0	0.0	0.0	0.0
tree-exa10-15	2	30	28	29	0.1	0.0	0.0	0.0
tree-exa10-20	2	40	38	39	0.2	0.0	0.0	0.1
tree-exa10-25	2	50	48	49	0.3	0.0	0.0	0.1
tree-exa10-30	2	60	58	59	0.5	0.0	0.0	0.1
z4ml.blif_0.10_0.20_0_1_inp_exact	5	65	193	1087	1759.2	0.7	1.3	2.9
z4ml.blif_0.10_0.20_0_1_out_exact	3	61	185	1221	2480.7	0.8	1.5	3.3
z4ml.blif_0.10_1.00_0_1_inp_exact	3	66	200	546	155.1	0.3	0.3	0.9
z4ml.blif_0.10_1.00_0_1_out_exact	3	64	196	1219	2411.1	0.8	1.5	3.3

Table 3. Evaluation results for various time limits

time limit (s)	success rate (%)	average time (s)	quantifier blocks	variables	clauses
5	33	0.9	41	286	649
60	53	12	53	1378	5458
600	81	73	133	3015	17752
3000	94	248	133	11570	19663

success rate is 94 percents for the time limit of 3000 seconds. Also one can see that we are able to solve instances with thousands of variables.

We have decided to show detailed evaluation results for the time limit of 60 seconds. We present our data in the same format as Weber [24] to allow easy comparison of the results. The data can be found in Table 2. The first column contains the name of a benchmark; the next three columns give a characterization of a formula by providing three size parameters of the formula – the number of the quantifier blocks, variables and clauses. The fifth column gives the size of Squolem's certificate measured by the number of the generated extensions. The next column contains the run-time of our system without the alpha-equivalence optimization. The symbol ∞ denotes the case when the time limit of 3600 seconds was exceeded.

The last three columns contain run-times for validation of Squolem's certificates in HOL Light. The SAT column tells how much time we spent by constructing the proof of (3) using the external SAT solver, i.e., by validating the model (see 4.2). The model column shows the run-time of proving the quantified model term (see 4.4). The last column finally contains the total run-time of our system for the given problem. All run-time columns are given in seconds and rounded to one decimal place. If we consider only instances for which we have data for non-optimized implementation, we get a speed-up factor of 321.0 for non-optimized vs. optimized implementation.

6 Conclusions and Future Work

We have developed and implemented a system that constructs proofs of valid QBFs from Squolem's certificates of validity. Our evaluation showed that this task is feasible – more than half of our benchmarks were solved within the time limit of 60 seconds. We had a similar experience with implementation as in [24,25], namely that performance is very sensitive to used inferences and to implementation details of the inference kernel. We proposed an optimization in HOL Light's kernel concerning computation of alpha-equivalence of terms, and got a speed-up factor of 321.0. Our implementation is freely available from the following web address: `http://ktiml.mff.cuni.cz/~kuncar/squolem2hollight`.

As was discussed in detail in Section 1, our system has two main applications. First, our system increases the amount of automation of HOL Light, and allows HOL Light's users to prove QBFs that are beyond the scope of the built-in tactics of HOL Light. Proving these formulas without our work would demand considerable human effort. Second, our approach can be used for validating correctness of Squolem's results because of HOL Light's small LCF-style kernel. A small bug was found and resolved in Squolem due to our work.

An alternative approach to using the LCF-style kernel directly is the use of reflection. This alternative approach requires implementation and a proof of correctness of a checker for Squolem's certificates in the prover's logic. Then this checker is run without producing any proof. In general, reflection provides better performance and still relatively high correctness assurances. To our best knowledge, there has not been done any work on reflectively verifying QBF solvers. There is also no support for reflection in HOL Light, namely one would have to integrate a reflection rule into HOL Light's kernel allowing it to trust the results of such a verified checker.

Some possible directions for future work are as follows: (i) There is still small room for further optimization, but probably not so radical as we presented. (ii) It is possible to implement our approach in other LCF-style interactive theorem provers, namely HOL4 and Isabelle/HOL. One can expect that implementation can differ because of variations in their kernels. (iii) Another direction is to continue in the general research of automation of interactive theorem provers, and integrate other systems. Integration of the system MetiTarski [1] seems to be the next challenging research task.

Acknowledgments. The author would like to thank John Harrison for proposing various optimizations in the code. This research was partially supported by SVV project number 263 314.

References

1. Akbarpour, B., Paulson, L.C.: MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. J. Autom. Reasoning 44(3), 175–205 (2010)
2. Benedetti, M., Mangassarian, H.: QBF-Based Formal Verification: Experience and Perspectives, vol. 5, pp. 133–191 (2008)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. 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)
5. Kleine Büning, H., Zhao, X.: On Models for Quantified Boolean Formulas. In: Lenski, W. (ed.) Logic versus Approximation. LNCS, vol. 3075, pp. 18–32. Springer, Heidelberg (2004)
6. Dershowitz, N., Hanna, Z., Katz, J.: Bounded Model Checking with QBF. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 408–414. Springer, Heidelberg (2005)
7. Giunchiglia, E., Narizzano, M., Tacchella, A.: QBF Reasoning on Real-World Instances. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 105–121. Springer, Heidelberg (2005)
8. Gordon, M.: From LCF to HOL: a short history.. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction, pp. 169–186. MIT Press, Cambridge (2000)
9. Harrison, J.: Binary Decision Diagrams as a HOL Derived Rule. Comput. J. 38(2), 162–170 (1995)
10. Harrison, J.: Towards self-verification of HOL light. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 177–191. Springer, Heidelberg (2006)
11. Harrison, J.: The HOL Light theorem prover (2010), `http://www.cl.cam.ac.uk/~jrh13/hol-light/`

12. Harrison, J., Slind, K., Arthan, R.D.: HOL. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. LNCS (LNAI), vol. 3600, pp. 11–19. Springer, Heidelberg (2006)
13. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. Journal of Automated Reasoning 21, 279–294 (1998)
14. HOL contributors: HOL4 Kananaskis 6 source code (2010), `http://hol.sourceforge.net` (retreived February 6, 2011)
15. Hurd, J.: An LCF-Style Interface between HOL and First-Order Logic. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 134–138. Springer, Heidelberg (2002)
16. Hurd, J.: First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In: Archer, M., Vito, B.D., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), pp. 56–68, No. NASA/CP-2003-212448 in NASA Technical Reports (September 2003)
17. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A First Step Towards a Unified Proof Checker for QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 201–214. Springer, Heidelberg (2007)
18. Kröning, D., Wintersteiger, C.: A file format for QBF certificates (2007), `http://www.cprover.org/qbv/download/qbcformat.pdf` (retreived February 6, 2011)
19. Meng, J., Paulson, L.C.: Translating Higher-Order Clauses to First-Order Clauses. J. Autom. Reasoning 40(1), 35–60 (2008)
20. Meyer, A., Stockmeyer, L.: Word Problems Requiring Exponential Time. In: Proc. 5th ACM Symp. on the Theory of Computing, pp. 1–9 (1973)
21. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and certifying QBFs: A comparison of state-of-the-art tools. AI Commun. 22(4), 191–210 (2009)
22. Otwell, C., Remshagen, A., Truemper, K.: An Effective QBF Solver for Planning Problems.. In: Arabnia, H.R., Joshua, R., Ajwa, I.A., Gravvanis, G.A. (eds.) MSV/AMCS, pp. 311–316. CSREA Press, Boca Raton (2004)
23. 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)
24. Weber, T.: Validating QBF Invalidity in HOL4. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 466–480. Springer, Heidelberg (2010)
25. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2009); special Issue: Empirically Successful Computerized Reasoning

Applying ACL2 to the Formalization of Algebraic Topology: Simplicial Polynomials*

L. Lambán[1], F.J. Martín–Mateos[2], J. Rubio[1], and J.L. Ruiz–Reina[2]

[1] Dept. of Mathematics and Computation, University of La Rioja
Edificio Vives, Luis de Ulloa s/n. 26004 Logroño, Spain
{lalamban,julio.rubio}@unirioja.es
[2] Computational Logic Group
Dept. of Computer Science and Artificial Intelligence, University of Seville
E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
{fjesus,jruiz}@us.es

Abstract. In this paper we present a complete formalization, using the ACL2 theorem prover, of the *Normalization Theorem*, a result in Algebraic Simplicial Topology stating that there exists a homotopy equivalence between the chain complex of a simplicial set, and a smaller chain complex for the same simplicial set, called the normalized chain complex. The interest of this work stems from three sources. First, the normalization theorem is the basis for some design decisions in the Kenzo computer algebra system, a program for computing in Algebraic Topology. Second, our proof of the theorem is new and shows the correctness of some formulas found experimentally, giving explicit expressions for the above-mentioned homotopy equivalence. And third, it demonstrates that the ACL2 theorem prover can be effectively used to formalize mathematics, even in areas where higher-order tools could be thought to be more appropriate.

1 Introduction

The origin of this work is a Computer Algebra system called *Kenzo* [2]. It is a Common Lisp program created by F. Sergeraert around 1990 and devoted to computing homology groups of topological spaces. In other words, Kenzo is a system devoted to *Algebraic Topology*. The goal of Algebraic Topology is to classify or to distinguish topological spaces by observing some algebraic structures associated with them.

Kenzo has been able to compute relevant results in the field, which have not been confirmed or refuted by any other means (see [11]). This is the reason why it makes sense to apply formal methods to study Kenzo and its correctness as a software system. And when talking about mechanized theorem proving and Kenzo, it is natural to think about ACL2 [4], a first-order theorem prover for reasoning about programs written in an extension of an applicative subset of Common

* Partially supported by Ministerio de Ciencia e Innovación, project MTM2009-13842, and by European Commission FP7, STREP project ForMath.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 200–215, 2011.

Lisp. Although Kenzo is not programmed in such an applicative subset, we could increase our confidence in some fragments of the Kenzo code, by formally verifying applicative (and executable) versions very closely related to the original code. Some preliminary results following this line have already been obtained [7,3].

In this paper, instead of verifying a fragment of the Kenzo code, we study a different aspect of the problem: since the underlying mathematical theory in the algorithms implemented in Kenzo is Algebraic Topology, we will have to formalize in ACL2 the main theorems on which Kenzo is based. Here we present a first step in this task: we show the ACL2 proof of a fundamental result in Algebraic Topology, the so-called *Normalization Theorem* [5]. As we will explain, this theorem is like a precondition for Kenzo, allowing it to deal with simpler structures.

It turns out that the ACL2 first-order logic is enough to prove this theorem. A symbolic setting is introduced in which the theorem can be proved by using only simplification and induction on lists, the kind of proofs ACL2 was designed for. Thus, this work could be considered a first milestone to formalize algebraic topology in a first order framework.

The organization of the paper is as follows. In Section 2 we introduce the minimal mathematical machinery needed to state and understand the main theorem proved. In Section 3, we present the ACL2 definitions and theorems formally establishing the result. In Section 4, we describe the symbolic framework of *simplicial polynomials*, a fundamental tool for the development of the proof. The paper ends with a section of conclusions and further work.

In our description of the formalization, we will necessarily skip many details and some of the function definitions will be omitted. The complete source files containing the ACL2 formalization and proof of the Normalization Theorem are accessible at `http://www.glc.us.es/fmartin/acl2/wfoe`.

2 Algebraic Simplicial Topology

In this section the most important concepts needed to state the main theorem are presented. More details can be found, for instance, in [8].

Definition 1. *A* simplicial set *K is a graded set $\{K_n\}_{n\in\mathbb{N}}$ together with functions:*

$$\begin{array}{llll} \partial_i^n : K_n \to K_{n-1}, & n > 0, & i = 0, \ldots, n, \\ \eta_i^n : K_n \to K_{n+1}, & n \geq 0, & i = 0, \ldots, n, \end{array}$$

subject to the following equations (called simplicial identities*):*

$$\begin{array}{lcccc} (1)\ \partial_i^{n-1}\partial_j^n & = & \partial_j^{n-1}\partial_{i+1}^n & \textit{if} & i \geq j, \\ (2)\ \eta_i^{n+1}\eta_j^n & = & \eta_{j+1}^{n+1}\eta_i^n & \textit{if} & i \leq j, \\ (3)\ \partial_i^{n+1}\eta_j^n & = & \eta_{j-1}^{n-1}\partial_i^n & \textit{if} & i < j, \\ (4)\ \partial_i^{n+1}\eta_j^n & = & \eta_j^{n-1}\partial_{i-1}^n & \textit{if} & i > j+1, \\ (5)\ \partial_i^{n+1}\eta_i^n & = & \partial_{i+1}^{n+1}\eta_i^n & = & id^n, \end{array}$$

where id^n denotes the identity function on K_n.

The functions ∂_i^n and η_i^n are called *face* and *degeneracy* maps, respectively. The elements of K_n are called *n-simplexes* (or simplexes of *dimension n*).

A simplicial set is a combinatorial model of a topological space and n-simplexes can be seen as an abstraction (and a generalization to dimension n) of the notion of triangle, given by its vertices. Although we have do not have enough room here to illustrate the notion of simplicial set, we get some intuition if we give one concrete simplicial set: think of n-simplexes as non-decreasing integer lists of length $n+1$ and interpret ∂_i^n and η_i^n as the functions that respectively delete and duplicate the i-th element of a list. This simplicial set is a particular case of a *simplicial complex* [1]. The notion of simplicial set is an *abstraction* of a simplicial complex, where simplexes are no longer lists, but whatever elements, where the simplicial identities hold.

If no confusion can arise, usually we remove the superindex in the face and degeneracy maps, writing simply ∂_i and η_i, respectively.

Algebraic Topology associates algebraic objects to topological spaces, and in particular to simplicial sets. To understand this precisely, we need some algebraic notions. A *chain complex* C is a sequence of pairs $\{C_n, d_n\}_{n\in\mathbb{N}}$, where each C_n is an abelian group and each d_n is a homomorphism $d_n : C_n \to C_{n-1}$ (called *boundary map* or *differential*) such that $d_n \circ d_{n+1} = 0$. This last property is called the *boundary property*, and can be restated as $Im(d_{n+1}) \subseteq Ker(d_n)$. Therefore, it is possible to consider the quotient group $Ker(d_n)/Im(d_{n+1})$, which is called the n-th *homology group* of the chain complex C, denoted $H_n(C)$.

Given a simplicial set K, we can associate to it some homology groups in the following way. For each $n \in \mathbb{N}$, let us consider the free abelian group generated by the n-simplexes K_n, group denoted by $C_n(K)$. That is, the elements of such a group are formal linear combinations $\sum_{j=1}^{r} \lambda_j x_j$, where $\lambda_j \in \mathbb{Z}$ and $x_j \in K_n$. These linear combinations are called *chains of simplexes* or, in short, *chains*. We define the homomorphisms $d_n : C_n(K) \to C_{n-1}(K)$ first defining them over each generator: for each $x \in K_n$, define $d_n(x) = \sum_{i=0}^{n} (-1)^i \partial_i(x)$; we then extend them to chains by linearity. It can be proved, using the simplicial identity (1), that these homomorphisms have the boundary property, and thus we say that the family of pairs $\{(C_n(K), d_n)\}_{n\in\mathbb{N}}$ is the chain complex associated to the simplicial set K, denoted by $C(K)$. Its homology groups are denoted by $H_n(K)$. Much effort is devoted in Algebraic Topology to studying and determining such homology groups, since it can be proved that they provide topological information that aids in the classification of spaces. Homology groups are the main objects to be computed by Kenzo.

There is an alternative way to associate a chain complex to a simplicial set K, taking into account only non-degenerate simplexes. We say that a n-simplex is *degenerate* if it is the result of applying a degeneracy map to a $n-1$ simplex; otherwise, it is *non-degenerate*. Let us denote by K_n^{ND} the set of non-degenerate n-simplexes of K, and $C_n^N(K)$ the free abelian group $\mathbb{Z}[K_n^{ND}]$ generated by non-degenerate simplexes. To get an actual chain complex, we introduce a differential map d_n^N which is defined as applying d_n and then erasing, in its image, the generators which are degenerate.

We define a family f of canonical epimorhisms $f_n : C_n(K) \to C_n^N(K)$ such that $f_n(\sum_{j=1}^{r} \lambda_j x_j)$ consists simply of eliminating all the addends $\lambda_j x_j$ such that x_j is a degenerate simplex. Note that the map f is compatible with respect to the differentials; that is to say, $f_{n-1} \circ d_n = d_n^N \circ f_n$. A function with this property is called a *chain morphism*.

The main property of the above canonical chain morphism f is that it preserves the homological information associated to a simplicial set, and this is established by the Normalization Theorem:

Theorem 1 *(Normalization Theorem). Let K be a simplicial set. Then the canonical homomorphism $f : C(K) \to C^N(K)$ induces group isomorphisms $H_n(C(K)) \cong H_n(C^N(K)), \forall n \in \mathbb{N}$.*

The theorem explains that, from the computational point of view, it is the same to work with $C(K)$ as with $C^N(K)$. This justifies a fundamental implementation decision in the Kenzo system: work only with the smaller chain complex $C^N(K)$ to compute homology groups of a simplicial set K.

A proof of the Normalization Theorem can be found, for example, in [5] (pages 236-237). Nevertheless, we will prove the result trying a stronger and more direct approach, more suitable for the ACL2 logic. This approach is based on the notions of *strong homotopy equivalence* and *reduction*:

Definition 2. *A* strong homotopy equivalence *is a 5-tuple (C, C', f, g, h)*

$$\begin{tikzcd} C \arrow[loop left, "h"] \arrow[r, bend left, "f"] & C' \arrow[l, bend left, "g"] \end{tikzcd}$$

where $C = (M, d)$ and $C' = (M', d')$ are chain complexes, $f\colon C \to C'$ and $g\colon C' \to C$ are chain morphisms, $h = (h_i\colon M_i \to M_{i+1})_{i\in\mathbb{N}}$ is a family of homomorphisms (called a homotopy operator*), which satisfy the following three properties for all $i \in \mathbb{N}$:*

(1) $f_i \circ g_i = id_{M'_i}$
(2) $d_{i+2} \circ h_{i+1} + h_i \circ d_{i+1} + g_{i+1} \circ f_{i+1} = id_{M_{i+1}}$
(3) $f_{i+1} \circ h_i = 0$

If, in addition the 5-tuple satisfies the following two properties:

(4) $h_i \circ g_i = 0$
(5) $h_{i+1} \circ h_i = 0$

then we say that it is a reduction*.*

This concept precisely describes a situation where the homological information is preserved. More concretely, if (C, C', f, g, h) is a reduction, then f_n induces an isomorphism of groups (with g_n defining the corresponding inverse) between $H_n(C)$ and $H_n(C'), \forall n > 0$. Therefore the following statement describes a stronger version of the Normalization Theorem:

Theorem 2 *(Normalization Theorem, reduction version). For every simplicial set K, there exists a reduction $(C(K), C^N(K), f, g, h)$ where f is the canonical chain epimorphism.*

An explicit definition of a possible reduction for this theorem was presented in [10] as an experimental result. There, after running several examples, it was conjectured (but not proved) that some possible formulas for the functions g and h could be:

- $g_m = \sum (-1)^{\sum_{i=1}^{p} a_i + b_i} \eta_{a_p} \dots \eta_{a_1} \partial_{b_1} \dots \partial_{b_p}$, where the indexes range over $0 \leq a_1 < b_1 < \dots < a_p < b_p \leq m$, with $0 \leq p \leq (m+1)/2$.
- $h_m = \sum (-1)^{a_{p+1} + \sum_{i=1}^{p} a_i + b_i} \eta_{a_{p+1}} \eta_{a_p} \dots \eta_{a_1} \partial_{b_1} \dots \partial_{b_p}$, where the indexes range over $0 \leq a_1 < b_1 < \dots < a_p < a_{p+1} \leq b_p \leq m$, with $0 \leq p \leq (m+1)/2$.

We have proved in ACL2 that, with these formulas for g and h, we have a strong homotopy equivalence. That was the most difficult part of all our formalization (note the complexity of the definitions above, which are very combinatorial). After proving that, we applied some general transformations to the function h, in such a way that we get properties (4) and (5), while preserving properties (1), (2) and (3). That is, we proved Theorem 2 in ACL2.

3 The Normalization Theorem in ACL2

In this section, we show the main definitions and theorems formalizing the Normalization Theorem in ACL2. We will leave for the next section a description of the main aspects of the proof.

Although the syntax of ACL2 terms and formulas is that of Common Lisp, and thus they are written using prefix notation, for the sake of readability they will be presented using a notation closer to the usual mathematical notation than its original Common Lisp syntax. For example, some of the functions will be presented in infix notation. When needed, we will show the correspondence between the ACL2 functions and the mathematical notation used instead.

3.1 Simplicial Sets and Chain Complexes

The first step in our formalization is the definition of the notion of simplicial set, as presented in Definition 1. Since the theorem we want to prove is a result on any simplicial set, we introduce a generic simplicial set using the encapsulation principle. In ACL2, `encapsulate` allows us to introduce functions in a consistent way, without giving a complete definition and only assuming certain properties about them.

A simplicial set can be defined by means of three functions `K`, `d` and `n`. The function `K` is a predicate with two arguments, such that `K`(m,x) is intended to mean $x \in K_m$. The functions `d` and `n` both have three arguments and they represent the face and degeneracy maps, respectively. The intended meanings for `d`(m,i,x) and `n`(m,i,x) are respectively $\partial_i^m(x)$ and $\eta_i^m(x)$. To be generic, we introduce them using the encapsulation principle: the only assumed properties about `K`, `d` and `n` are those stating well-definedness of `d` and `n` and the five simplicial identities. We do not list here all those properties, but for example these are the assumptions about the well-definedness of the face map, and the first simplicial identity:

Assumption: `d-well-defined`
$(x \in K_m \wedge m \in \mathbb{N}^+ \wedge i \in \mathbb{N} \wedge i \leq m) \rightarrow \partial_i^m(x) \in K_{m-1}$
Assumption: `simplicial-id1`
$(x \in K_m \wedge m \in \mathbb{N} \wedge i \in \mathbb{N} \wedge j \in \mathbb{N} \wedge j \leq i \wedge i < m \wedge 1 < m)$
$\rightarrow \partial_i^{m-1}(\partial_j^m(x)) = \partial_j^{m-1}(\partial_{i+1}^m(x))$

The next step is to define chain complexes. Since chains are linear combinations of simplexes of a given dimension, it is natural to represent them as lists whose elements are (dotted) pairs formed by an integer and a simplex. We will consider only chains in canonical form: their elements must have non-null coefficients and have to be strictly increasingly ordered with respect to a total order (given by the function `ss-<`, which is based on the ACL2 primitive function `lexorder`). The main advantage of this is that the equality between chains will simply be the ACL2 syntactical equality (`equal`).

The following function `sc-p` defines chains in a given dimension (the auxiliary function `ss-p` defines the dotted pairs formed by a non-null integer and a simplex of a given dimension):

Definition:

```
sc-p($m$,$c$) :=
    if endp($c$) then $c$ = nil
    elseif endp(cdr($c$)) then ss-p($m$,first($c$)) $\wedge$ rest($c$) = nil
    else ss-p($m$,first($c$)) $\wedge$ ss-<($m$,first($c$),second($c$)) $\wedge$
          sc-p($m$,rest($c$))
```

The main operations we define on chains are addition and scalar product by an integer, for each dimension m. The ACL2 functions for these operations are `add-sc-sc`(m,c_1,c_2) and `scl-prd-sc`(m,k,c), whose definition we omit here. Recall that we have to take care of returning their result in canonical form. From now on, we will respectively denote them as $c_1 + c_2$ and $k \cdot c$ (note that, for the sake of readability, we omit the dimension).

The set of chains of a given dimension is an abelian group with respect to addition, where the identity in this group is the zero chain (represented as `nil` and denoted here as 0). It is worth mentioning that we automatically obtained all the definitions and theorems proving the group structure of chains, as an instance of a more generic theory about the free abelian group generated by a generic basis. For that automatic generation we used the generic instantiation tool described in [6].

Simplicial maps can be linearly extended on chains. For example, this is the definition of c-d, the face map extended to chains[1]:

```
DEFINITION: [$\partial_i^m(c)$]
  c-d($m$,$i$,$c$) :=
    if endp($c$) then $c$
    else cons(car(first($c$)),$\partial_i^m$(cdr(first($c$)))) + c-d($m$,$i$,rest($c$)))
```

Note that this function is not a simple "mapcar" on the simplexes of a chain, since its result is returned in canonical form. In a similar way, we define c-n, the extension of the degeneracy map to chains. We will use the same notation ($\partial_i^m(c)$ and $\eta_i^m(c)$) to denote these maps both on simplexes and on chains.

Let us now define the differential on chains. Recall that its precise definition is $d_m(c) = \sum_{i=0}^{m}(-1)^i \partial_i^m(c)$. The following function diff implements the corresponding ACL2 recursive definition (the auxiliary function diff-aux is needed to introduce an extra argument n for the dimension on where the function is defined, which remains unchanged during the recursion):

```
DEFINITION:
  diff-aux($n$,$m$,$c$) :=
    if $m \notin \mathbb{N}^+$ then $\partial_0^n(c)$
    else $(-1)^m \cdot \partial_m^n(c)$ + diff-aux($n$,$m-1$,$c$))
DEFINITION: [$d_m(c)$]
  diff($m$,$c$) := diff-aux($m$,$m$,$c$)
```

The following theorem states that the above function satisfies the boundary property, and thus we have a chain complex:

THEOREM: diff-diff=0

$$(m \in \mathbb{N}^+ \wedge c \in C_{m+1}(K)) \rightarrow d_m(d_{m+1}(c)) = 0$$

3.2 The Normalized Chain Complex

We now describe the formalization of the normalized chain complex $C^N(K)$. First of all we define degenerate simplexes (those that can be obtained applying a degeneracy map to another simplex) and the complementary set of non-degenerate simplexes:

```
DEFINITION: [$x \in K_m^D$]
  Kd($m$,$x$) := $\exists y,i\ (i \in \mathbb{N} \wedge i < m \wedge y \in K_{m-1} \wedge \eta_i^{m-1}(y) = x)$
DEFINITION: [$x \in K_m^{ND}$]
  Kn($m$,$x$) := $x \in K_m \wedge x \notin K_m^D$
```

[1] Note the expression between square brackets in the first line of the definition of the function. In general, this is the way we will present the notation subsequently used in the paper for a defined function, when it is different from the actual ACL2 prefix notation in the sources.

The existential quantifier in the definition of K^D_m is introduced by `defun-sk`, which is the way ACL2 provides support for first-order quantification.

Since normalized chains are linear combinations of non-degenerate simplexes of a given dimension, we represent them in the same way as we represent general chains, but in this case requiring non-degenerate generators. As with general chains, the definitions and theorems corresponding to the group properties (w.r.t. addition) of the set of normalized chains $C^N_m(K)$, are obtained automatically as an instance of the generic theory of freely generated groups (again using the generic instantiation tool [6]).

We also proved that it is a subgroup of $C_m(K)$ so it makes sense to denote $c_1 + c_2$ and $k \cdot c$ the addition and scalar product of normalized chains. In general, any function on chains can be also applied to normalized chains.

We define the canonical epimorphism $f : C(K) \to C^N(K)$ simply as the function that, given an element of $C_m(K)$, returns the normalized chain obtained by eliminating its degenerate addends. In our formalization, the following function `F-norm` defines f (here `ssn-p` checks the property of being a non-degenerate addend, and it uses the function `Kn` above):

Definition: $[f_m(c)]$

```
F-norm($m$,$c$) :=
    if endp($c$) then 0
    elseif ssn-p($m$,first($c$))
      then first($c$) + F-norm($m$,rest($c$)))
    else F-norm($m$,rest($c$))
```

A key property relating the canonical chain epimorphism f and the differential on $C(K)$ is the following: if we apply normalization on the result of the differential of a chain, we obtain the same result as if we apply the same operation previously normalizing the chain. This is a consequence of the simplicial identities and it is established by the following theorem:

Theorem: `diff-n-F-norm`
$(m \in \mathbb{N}^+ \wedge c \in C_m(K)) \to f_{m-1}(d_m(f_m(c))) = f_{m-1}(d_m(c))$

The differential operation of the normalized chain complex $C^N(K)$, denoted as $d^N_m(c)$, is defined as the result of applying the differential d_m, and after that, normalizing with f_{m-1}:

Definition: $[d^N_m(c)]$
`diff-n`$(m,c) := f_{m-1}(d_m(c))$

The differential property for d in $C(K)$ (theorem `diff-diff=0` in the previous subsection), together with the property `diff-n-F-norm`, allows us to prove the differential property for d^N in $C^N(K)$, since for all $c \in C^N_m(K)$, $d^N_m(d^N_{m+1}(c)) = f_{m-1}(d_m(f_m(d_{m+1}(c)))) = f_{m-1}(d_m(d_{m+1}(c))) = f_{m-1}(0) = 0$. The following theorem establishes it:

Theorem: `diff-n-diff-n=0`
$(m \in \mathbb{N}^+ \wedge c \in C^N_{m+1}(K)) \to d^N_m(d^N_{m+1}(c)) = 0$

3.3 Defining the Reduction

Once f is defined, it remains to define the functions g and h needed for the reduction version of the Normalization Theorem. As we have said, our definitions are based on the formulas experimentally conjectured in [10], presented at the end of Section 2. The following function G is a recursive version of the formula for g_m (again we need an auxiliary function for dealing properly with the dimension):

```
DEFINITION:
  G-aux($n$,$m$,$c$) :=
    if $m \notin \mathbb{N}^+$ then $c$
    else G-aux($n$,$m-1$,$c - \eta^{n-1}_{m-1}(\partial^n_m(c))$)
DEFINITION: [$g_m(c)$]
  G($m$,$c$) := G-aux($m$,$m$,$c$)
```

Some explanation is needed, to give an intuitive idea of why this recursive version implements the explicit formula for g_m of Section 2. Note that the terms in that explicit formula are of two types: those not containing ∂_m, which are precisely the terms of g_{m-1}, and those containing ∂_m, which can be obtained composing g_{m-1} with $\eta_{m-1}\partial_m$, and then applying the simplicial identities.

Now we define the function H0, the recursive version of the formula for h_m conjectured in [10] (the reason why we call it H0 instead of H will be clear soon). For this definition, we need to define auxiliaries functions A-aux and H0-aux:

```
DEFINITION:
  A-aux($n$,$m$,$c$) :=
    if $m \notin \mathbb{N}^+$ then $0$
    else $-$A-aux($n$,$m-1$,$\eta^{n-1}_{m-1}(\partial^n_m(c))$) +
        $(-1)^{m-1} \cdot \eta^n_m($G-aux($n$,$m-1$,$\eta^{n-1}_{m-1}(\partial^n_m(c))$))
DEFINITION:
  H0-aux($n$,$m$,$c$) :=
    if $m \notin \mathbb{N}^+$ then $\eta^n_0(c)$
    else H0-aux($n$,$m-1$,$c$) + $(-1)^m \cdot \eta^n_m(c)$ + A-aux($n$,$m$,$c$)
DEFINITION: [$h^0_m(c)$]
  H0($m$,$c$) := H0-aux($m$,$m$,$c$)
```

An intuitive idea of why this recursive definition is equivalent to the explicit definition for h_m given in Section 2, is the following. Again, the terms in that explicit definition are of two types, depending on whether they contain ∂_m or not. Those not containing ∂_m are precisely the terms in $h_{m-1} + (-1)^m \cdot \eta_m$. Now, the idea introducing a_m (i.e., A-aux) is to generate all the terms of h_m containing ∂_m. To see this, note that these terms can be, in turn, of two types, depending on whether they do not contain η_m or they do. In the first case, these terms can be obtained composing $-a_{m-1}$ and $\eta_{m-1}\partial_m$. In the second case, these terms can be obtained composing η_m with every term in g_m containing ∂_m. And the terms in g_m containing ∂_m are obtained composing g_{m-1} and $\eta_{m-1}\partial_m$. Note again that we need to apply the simplicial identities to get the face and degeneracy maps composed in the same order as they are in the explicit formula.

We realized, in the course of our proof attempt, that with this definition for h^0, we only have a strong homotopy equivalence. Fortunately, it is possible to obtain, with a two-step transformation, a function h_m from h^0_m, having properties (4) and (5) and still preserving the homotopy equivalence properties. The following defines `H` by a two-step transformation from `H0` (it turns out that with the first transformation, we get (4) and with the second we get (5)):

Definition: $[h^1_m(c)]$
`H1`$(m,c) := h^0_m(c) - h^0_m(g_m(f_m(c)))$
Definition: $[h_m(c)]$
`H`$(m,c) := h^1_m(d_{m+1}(h^1_m(c)))$

3.4 The Main Theorems

Now that we have defined the 5-tuple $(C(K), C^N(K), f, g, h)$, we present here the main theorems proved, showing that it is a reduction:

Theorem: `F-chain-morphism`
$(m \in \mathbb{N}^+ \wedge c \in C_m(K)) \rightarrow d^N_m(f_m(c)) = f_{m-1}(d_m(c))$
Theorem: `G-chain-morphism`
$(m \in \mathbb{N}^+ \wedge c \in C^N_m(K)) \rightarrow g_{m-1}(d^N_m(c)) = d_m(g_m(c))$
Theorem: `F-G-H-property-1`
$(m \in \mathbb{N} \wedge c \in C^N_m(K)) \rightarrow f_m(g_m(c)) = c$
Theorem: `F-G-H-property-2`
$(m \in \mathbb{N}^+ \wedge c \in C_m(K)) \rightarrow d_{m+1}(h_m(c)) + h_{m-1}(d_m(c)) = c - g_m(f_m(c))$
Theorem: `F-G-H-property-3`
$(m \in \mathbb{N} \wedge c \in C_m(K)) \rightarrow f_{m+1}(h_m(c)) = 0$
Theorem: `F-G-H-property-4`
$(m \in \mathbb{N} \wedge c \in C^N_m(K)) \rightarrow h_m(g_m(c)) = 0$
Theorem: `F-G-H-property-5`
$(m \in \mathbb{N} \wedge c \in C_m(K)) \rightarrow h_{m+1}(h_m(c)) = 0$

These properties establish in ACL2 the Normalization Theorem in its reduction version. In the following section, we describe the main aspects of the proof of these theorems. In particular, we present a framework where most of the reasoning was carried out: what we call the simplicial polynomial framework.

4 Simplicial Polynomials

Our ACL2 proof of the Normalization Theorem was developed following the usual interaction with the system. Based on a hand proof, we guided the prover, by means of a number of definitions and lemmas, suggested at a high level from the hand proof, and at a lower level from inspection of failed proof attempts. In this case, we also needed to do the hand proof on our own.

Roughly speaking, most of the proofs of the theorems of the previous section can be carried out by manipulating symbolic expressions that *represent* sums of compositions of face and degeneracy maps in a certain canonical way. These expressions are what we call *simplicial polynomials*. Moreover, most of the lemmas and theorems can be proved applying induction and equational reasoning on functions that return simplicial polynomials.

So our approach to get the proof of the Normalization Theorem was to define simplicial polynomials in ACL2 (using lists and numbers) and operations on them resembling addition and composition of functions. We then proved the properties showing that with respect to these operations, simplicial polynomials are a ring. Guided by our hand proofs, most of the results needed for the Normalization Theorem can be proved conveniently in the ring of simplicial polynomials. Finally, we "lifted" the theorems proved in the simplicial polynomial framework to the formalization presented in the previous section (which from now on will be referred to as the *standard formalization*).

Due to the lack of space, we prefer to concentrate on the description of the simplicial polynomial framework and how we used it as a convenient tool to get the mechanical proof of the Normalization Theorem. For details on the mathematical contents of the proof, we refer the reader to the sources.

4.1 The Ring of Simplicial Polynomials

Before describing simplicial polynomials, let us illustrate how we can represent any composition of face and degeneracy maps using only lists and numbers. Let $\partial_5^5\eta_3^4\partial_1^5\partial_2^6\eta_4^5$ be a composition of maps defined to act on chains of dimension 5. The first observation is that we can drop the superindexes, because once we know on which dimension the composition is defined, then the superindex of each map can be determined[2]. The second observation is that we can apply the simplicial identities as rewrite rules to transform the composition to an equivalent canonical form in which, from left to right, and with respect to their subindexes, there is a strictly decreasing sequence of degeneracy maps followed by an strictly increasing sequence of face maps. In our example, this equivalent canonical form is $\eta_3\eta_2\partial_1\partial_2\partial_5$, which can be represented by the two-element list `((3 2) (1 2 5))`.

A *simplicial term* is a list containing two lists of natural numbers, representing canonical compositions. The first list (representing the degeneracies) is strictly decreasing and the second (representing the faces) is strictly increasing. Since simplicial terms represent functions that are applied to chains, we also have to consider in our representation "sums" of simplicial terms, possibly with an integer coefficient. In this context, a *monomial* is defined to be a (dotted) pair of an integer and a simplicial term, and a *simplicial polynomial* is simply a list of monomials. For example, the expressions $\boldsymbol{p}_1 = 3 \cdot \eta_4\eta_1\partial_3\partial_6\partial_7 - 2 \cdot \eta_1\partial_3\partial_4$ and $\boldsymbol{p}_2 = \eta_3\partial_4\partial_6 + 2 \cdot \eta_1\partial_3\partial_4$ are both simplicial polynomials (for the sake of clarity we maintain the $+$, η and ∂ symbols in the examples, but it has to be clear that simplicial polynomials are represented using only lists and numbers).

[2] Note that we cannot ignore the superindexes in the standard formalization of the theorem, since our goal is to do a precise formalization of the mathematical theory. What we will do now is to formally justify that we can prove most of the properties without explicitly including the superindexes.

As with chains, in our ACL2 representation we will only consider simplicial polynomials in canonical form: a true list of monomials, with non-null coefficients, and strictly increasingly ordered with respect to a fixed total order on simplicial terms. This allows us to check the equality of two simplicial polynomials by simply using the ACL2 syntactic equality `equal`. Thus, functional equality is reduced to syntactic equality of first-order objects.

We can define operations on simplicial polynomials corresponding to the addition and composition of the functions they represent. For example, the addition of $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ above is the polynomial $\eta_3\partial_4\partial_6 + 3 \cdot \eta_4\eta_1\partial_3\partial_6\partial_7$ and their composition is $-2 \cdot \eta_1\partial_3\partial_4\partial_6 - 4 \cdot \eta_2\eta_1\partial_2\partial_3\partial_4\partial_5 + 3 \cdot \eta_4\eta_1\partial_4\partial_6\partial_7\partial_8 + 6 \cdot \eta_4\eta_2\eta_1\partial_2\partial_3\partial_4\partial_7\partial_8$. Of course, there is trade-off with the clean treatment of the equality: it makes the definitions of operations between polynomials (and the proof of their properties) more difficult, since we have to return the results also in canonical form. For example, the definition of the composition of simplicial terms and the proof of its associativity turned out to be particularly difficult.

In our formalization, `sp-p`, denoted here as $\boldsymbol{p} \in \mathcal{P}$, recognizes those ACL2 objects that represent simplicial polynomials (in the following we will use boldface to denote polynomials). The functions `add-sp-sp` and `cmp-sp-sp`, whose definition we omit here, respectively implement addition and composition (or product) of simplicial polynomials, denoted respectively as $\boldsymbol{p}_1 + \boldsymbol{p}_2$[3], and $\boldsymbol{p}_1 \cdot \boldsymbol{p}_2$. An interesting by-product of using simplicial polynomials, unlike the standard formalization, is that operations are executable (particularly interesting for us, since our long-term goal is the verification of a symbolic computation system).

We proved the properties showing that with respect to these two operations, simplicial polynomials have a ring structure. For example, the following establishes right distributivity of composition with respect to addition:

Theorem: `cmp-sp-sp-add-sp-sp-distributive-r`

$$(\boldsymbol{p}_1 \in \mathcal{P} \wedge \boldsymbol{p}_2 \in \mathcal{P} \wedge \boldsymbol{p}_3 \in \mathcal{P}) \to \boldsymbol{p}_1 \cdot (\boldsymbol{p}_2 + \boldsymbol{p}_3) = (\boldsymbol{p}_1 \cdot \boldsymbol{p}_2) + (\boldsymbol{p}_1 \cdot \boldsymbol{p}_3)$$

We do not list here all the ring properties proved, and we refer the reader to the sources for a detailed description. All those properties are essential in our formalization, and extensively used in the proofs.

It is worth pointing out that we proved all the ring properties as (functional) instances of a more generic formalization. In the sources, the reader will find the development of a general theory about the ring of linear combinations (with integer coefficients) of elements of a generic monoid (a set with an associative operation with identity). The ring of simplicial polynomials is just a particular instance of this generic theory (a related ACL2 development for polynomials in commutative algebra can be found in [9]). Specifically, we first proved that the set of simplicial terms is a monoid with respect to composition, and then the definitions of the operations on simplicial polynomials and their ring properties were automatically generated using the generic instantiation tool [6].

[3] For clarity, we are using the same symbol + that we used in the previous section for chain addition, but they are different operations.

4.2 Formal Proofs in the Polynomial Framework

Unfortunately, there is not a direct translation of the Normalization Theorem in the polynomial framework. The main reason is that the canonical epimorphism f (which we recall is defined deleting the degenerate simplexes of a chain), cannot be expressed as a simplicial polynomial. But fortunately, we can do most of the work (or at least, the hard part) at the polynomial level. The idea is to define polynomial versions for the differential d and for g and h, and prove, in the simplicial polynomial ring, their main properties.

For example, this is the definition of the polynomial that represents the function g_m introduced in Section 2. Here $\boldsymbol{id}$ is the ring identity with respect to composition (i.e., the polynomial representation of the identity function):

```
DEFINITION: $[\boldsymbol{g}_m]$
  G-pol($m$) :=
    if $m \notin \mathbb{N}^+$ then $\boldsymbol{id}$
    else G-pol($m-1$) $\cdot$ ($\boldsymbol{id} - \eta_{m-1}\partial_m$)
```

Note that this definition mimics, at a symbolic level, the recursive definition of g_m, but without the burden of dealing with superindexes and without explicitly giving the chain on which it is applied. In a similar way, we can define $\boldsymbol{d}_m$ and $\boldsymbol{h}_m$, the polynomial counterparts of the functions d_m and h_m.

Once defined these functions, we prove a number of lemmas about them, polynomial versions of the results we need to prove Theorem 2. For example, this is the polynomial version of the theorem stating that g_m is a chain morphism:

THEOREM: `G-pol-and-diff-pol-commute`
$m \in \mathbb{N} \rightarrow \boldsymbol{d}_m \cdot \boldsymbol{g}_m = \boldsymbol{g}_{m-1} \cdot \boldsymbol{d}_m$

All these properties, although with substantial differences in its difficulty, have been proved in a similar way: applying induction suggested by the recursive definitions and using the properties of the simplicial polynomial ring and the simplicial identities, to prove the inductive case. Again, we refer the reader to the source code, for details on the proofs.

4.3 Lifting Proofs from the Polynomial Framework

We now describe how we can translate the properties on polynomials, to the corresponding properties in the standard framework presented in Section 3. Roughly speaking, we can say that the "essence" of the property is already captured in the polynomial version, but some technical details have still to be solved when translating it: for example, the reintroduction of the superindexes or also how to incorporate the canonical epimorphism f in results that mention it.

The key (and natural) idea is to define the functional behavior of a simplicial polynomial by means of a function that receives a polynomial and a chain as input, and *evaluates* the polynomial on the chain by applying the maps and sums that it encodes. This function will also receive as input the expected dimension of the chain (this will allow us to properly reintroduce the superindexes).

To illustrate how we define this evaluation function, this is the definition of the auxiliary function used to evaluate a list of faces *ld* (the second element of a simplicial term) on a chain *c* of dimension *m*:

Definition:

```
eval-ld($ld$,$m$,$c$) :=
    if endp($ld$) then $c$
    else c-d($m$-len(rest($ld$)),first($ld$),eval-ld(rest($ld$),$m$,$c$)))
```

Recall that `c-d` (presented in Subsection 3.1) is the face map, linearly extended to chains; note also how the dimension is properly managed in the recursive call. In a similar way, we can define the evaluation of a list of degeneracies. Extending these, we define the evaluation of simplicial terms (`eval-st`), the evaluation of monomials (`eval-sm`) and finally the evaluation of a polynomial $\boldsymbol{p}$ on a chain c of dimension m, the function `eval-sp`($\boldsymbol{p}$,m,c).

Not every simplicial polynomial can be interpreted consistently as a function on chains. Think for example in the simplicial term $\eta_5\eta_2\eta_1\partial_1\partial_3$. It cannot be evaluated on chains of dimension less than 5, since otherwise in the last step we will be applying η_5 to a chain of dimension less than 5. In general, in the case that the simplicial term may be interpreted as a function on dimension m, we say that the simplicial term is *valid* for m. For example, the simplicial term of the example is valid for every dimension $m > 4$.

The *degree* of a simplicial term is an integer giving the "dimension jump" of every function it may represent (or equivalently, it is the difference between the number of degeneracies and the number of faces). It is clear that another restriction we must impose on a simplicial polynomial, in order to being able to evaluate it on chains, is that it has to be *uniform* (that is, all its terms have the same degree).

We have formalized those restrictions in ACL2 by means of three functions `valid-sp`, `uniform-sp` and `degree-sp`, whose definitions we omit here: `valid-sp` ($\boldsymbol{p}$,m) checks whether all the simplicial terms in $\boldsymbol{p}$ are valid for dimension m, `uniform-sp`($\boldsymbol{p}$) checks if all the terms in p have the same degree and `degree-sp`($\boldsymbol{p}$) is the common degree of the terms of a uniform polynomial (or 0 if it is the zero polynomial).

Now the fundamental properties of the evaluation function `eval-sp` are that for a given dimension, it behaves consistently with respect to the operations of the ring of simplicial polynomials, whenever the input polynomials are valid for that dimension and uniform. Note that these properties are not trivial, because again we have to deal with the canonical form.

Theorem: `eval-sp-add-sp-sp`

$(\boldsymbol{p_1} \in \mathcal{P} \land \boldsymbol{p_2} \in \mathcal{P} \land m \in \mathbb{N} \land c \in C_m(K) \land \texttt{valid-sp}(\boldsymbol{p_1},m) \land$
$\texttt{valid-sp}(\boldsymbol{p_2},m) \land \texttt{uniform-sp}(\boldsymbol{p_1}) \land \texttt{uniform-sp}(\boldsymbol{p_2}) \land$
$(\texttt{endp}(\boldsymbol{p_1}) \lor \texttt{endp}(\boldsymbol{p_2}) \lor \texttt{degree-sp}(\boldsymbol{p_1}) = \texttt{degree-sp}(\boldsymbol{p_2})))$
$\to \texttt{eval-sp}(\boldsymbol{p_1} + \boldsymbol{p_2},m,c) = \texttt{eval-sp}(\boldsymbol{p_1},m,c) + \texttt{eval-sp}(\boldsymbol{p_2},m,c))$

Theorem: `eval-sp-cmp-sp-sp`
$(\boldsymbol{p}_1 \in \mathcal{P} \wedge \boldsymbol{p}_2 \in \mathcal{P} \wedge m \in \mathbb{N} \wedge c \in C_m(K) \wedge \texttt{valid-sp}(\boldsymbol{p}_2,m) \wedge$
$\texttt{valid-sp}(\boldsymbol{p}_1,m+\texttt{degree-sp}(\boldsymbol{p}_2)) \wedge \texttt{uniform-sp}(\boldsymbol{p}_1) \wedge \texttt{uniform-sp}(\boldsymbol{p}_2))$
$\rightarrow \texttt{eval-sp}(\boldsymbol{p}_1 \cdot \boldsymbol{p}_2,m,c) =$
$\texttt{eval-sp}(\boldsymbol{p}_1,m+\texttt{degree-sp}(\boldsymbol{p}_2),\texttt{eval-sp}(\boldsymbol{p}_2,m,c))$

Now we can prove equivalences of the polynomial versions of the functions with their standard versions (since they are valid and uniform polynomials). For example, this is the result relating $\boldsymbol{g}_m$ and g_m (analogous theorems are proved for $\boldsymbol{d}_m$ and d_m, and for $\boldsymbol{h}_m$ and h_m):

Theorem: `G-eval-sp-G-pol`
$(m \in \mathbb{N} \wedge c \in C_m^N(K)) \rightarrow \texttt{eval-sp}(\boldsymbol{g}_m,m,c) = g_m(c)$

These properties allow us to directly translate the properties proved in the polynomial framework to the corresponding properties in the standard formalization. Let us illustrate this, for example, with the case of proving that g is chain morphism. From the property `G-pol-and-diff-pol-commute` at the end of the previous subsection, and using the equivalences proved, we obtain:

Theorem: `G-and-diff-commute`
$(m \in \mathbb{N}^+ \wedge c \in C_m(K)) \rightarrow g_{m-1}(d_m(c)) = d_m(g_m(c))$

This property is almost the property `G-chain-morphism`, one of the reduction properties needed for the Normalization Theorem. One last detail is missing, since in that property we mention the normalized differential d_m^N in the left hand side, instead of d_m. That is, we have to "incorporate" the canonical epimorphism to the theorem. But it is easy to prove that g_m applied to any degenerate simplex is 0, and therefore $g_m(f_m(c)) = g_m(c)$ for every chain c. This means that $g_{m-1}(d_m^N(c)) = g_{m-1}(f_{m-1}(d_m(c))) = g_{m-1}(d_m(c))$ and thus we finally obtain the theorem `G-chain-morphism`.

This example illustrate a typical situation in our formal proof. The main "combinatorial" property is proved at polynomial level (usually by induction), and then we use the equivalences and possibly some final easy simplifications to obtain the property in the standard framework.

5 Conclusions and Further Work

The work reported in this paper shows that the ACL2 theorem prover can be effectively used to mechanize non-trivial mathematics, in fields (like Algebraic Topology) where higher-order tools could be thought as more appropriate. Our case study is the *Normalization Theorem*, an important result in simplicial topology establishing a link between the two chain complexes that can be naturally associated to a simplicial set. As a by-product, our proof has been used to formally prove the correctness of some explicit formula experimentally found in [10].

To quantify the proof effort, the complete formalization contains 99 definitions and 565 lemmas and theorems (with 158 non-trivial proof hints explicitly given), which gives an idea of the degree of automation of the proof. It is worth pointing out that the whole development has benefited from the use of our instantiation tool for generic theories described in [6]. That allowed us to obtain in an automated way, the definitions and theorems proving the ring of simplicial polynomials and the abelian group of chains and normalized chains, as instances of generic theories (we have not included these automatically generated definitions and lemmas in the statistics).

Simplicial polynomials turned out to be a convenient tool for the proof of the Normalization Theorem, so our future work is to extend this technique to other problems in algebraic topology. Our next goal is the *Eilenberg-Zilber Theorem* [8]. It is a very important result giving a reduction between the chain complex of the cartesian product of simplicial sets, $C^N(A \times B)$, and the tensor product of the corresponding chain complexes of the factors, $C^N(A) \otimes C^N(B)$. The associated algorithm is very important in Kenzo, being responsible for most of the (exponential) complexity of many Kenzo programs. Thus the task of formalizing it can be considered a good next step. The challenge is that in the Eilenberg-Zilber Theorem there are two simplicial sets involved, and therefore the scope of our techniques should be significantly extended to be applied in that case.

References

1. De Loera, J.A., Rambau, J., Santos, F.: Triangulations. Structures for Algorithms and Applications. Springer, Heidelberg (2010)
2. Dousson, X., Sergeraert, F., Siret, Y.: The Kenzo Program. Institut Fourier, Grenoble (1999), `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`
3. Heras, J., Pascual, V., Rubio, J.: Proving with ACL2 the correctness of simplicial sets in the kenzo system. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 37–51. Springer, Heidelberg (2011)
4. Kaufmann, M., Manolios, P., Moore,J S.: Computer-Aided Reasoning: An Approach. Kluwer, Dordrecht (2000)
5. Mac Lane, S.: Homology. Springer, Heidelberg (1963)
6. Martín–Mateos, F.J., Alonso, J.A., Hidalgo, M.J., Ruiz–Reina, J.L.: A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. In: Proceedings of the Third International ACL2 Workshop and its Applications, pp. 188–201 (2002)
7. Martín-Mateos, F.J., Rubio, J., Ruiz-Reina, J.L.: ACL2 Verification of Simplicial Degeneracy Programs in the Kenzo System. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 106–121. Springer, Heidelberg (2009)
8. May, J.P.: Simplicial objects in Algebraic Topology. Van Nostrand, New York (1967)
9. Medina–Bulo, I., Palomo–Lozano, F., Ruiz–Reina, J.L.: A verified Common Lisp implementation of Buchberger's algorithm in ACL2. Journal of Symbolic Computation 45(1), 96–123 (2010)
10. Rubio, J., Sergeraert, F.: Supports Acycliques and Algorithmique. Astérisque 192, 35–55 (1990)
11. Rubio, J., Sergeraert, F.: Constructive Algebraic Topology. Bulletin Sciences Mathématiques 126, 389–412 (2002)

Animating the Formalised Semantics of a Java-Like Language

Andreas Lochbihler[1] and Lukas Bulwahn[2]

[1] Karlsruher Institut für Technologie
andreas.lochbihler@kit.edu
[2] Technische Universität München
bulwahn@in.tum.de

Abstract. Considerable effort has gone into the techniques of extracting executable code from formal specifications and animating them. We show how to apply these techniques to the large *JinjaThreads* formalisation. It models a substantial subset of multithreaded Java source and bytecode in Isabelle/HOL and focuses on proofs and modularity whereas code generation was of little concern in its design. Employing Isabelle's code generation facilities, we obtain a verified Java interpreter that is sufficiently efficient for running small Java programs. To this end, we present refined implementations for common notions such as the reflexive transitive closure and Russell's definite description operator. From our experience, we distill simple guidelines on how to develop future formalisations with executability in mind.

1 Introduction

In the last few years, substantial work has been devoted to the techniques and tools for executing formal specifications from Isabelle/HOL, on the levels of both prover infrastructure [5,8,9] and formalisations of foundational notions and concepts [6,11,18]. But so far, applications (e.g. [4,19,20]) have been designed for executability and restricted to purely functional specifications. A benchmark to test whether the aforementioned techniques mix well and scale to large formalisations has been missing.

In this work, we study how to apply code generation techniques to the *Jinja-Threads* project [15,16,17], which formalises a substantial subset of multithreaded Java source and bytecode. JinjaThreads constitutes a good benchmark for three reasons: (i) It is a large formalisation (70k lines of definitions and proofs) that involves a broad range of advanced Isabelle features. (ii) As a programming language, type system, and semantics, it has a built-in notion of execution. This sets the goal for what should be executable. (iii) It focuses on proofs and modularity rather than code generation, i.e. complications in specifications and proofs for the sake of direct code generation were out of the question. Hence, it tests if code generation works "in the wild" and not only for specialised developments.

Our main contribution here is to discuss what was needed to automatically generate a well-formedness checker and an interpreter for JinjaThreads programs

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 216–232, 2011.

from the Isabelle formalisation, and what the pitfalls were. Thereby, we demonstrate how to combine the different techniques and tools such that changes to the existing formalisation stay minimal. Our contributions fall into two parts.

On the system's side, we enhanced Isabelle's code generator for inductive predicates (§2.1) to obtain a mature tool for our needs. It now compiles inductive definitions and first-order predicates, interpreted as logic programs, to functional implementations. Furthermore, we present a practical method to overcome the poor integratability of Isabelle's code generator into Isabelle's module system (§2.2). Finally, we describe a tabled implementation of the reflexive transitive closure (§2.3) and an executable version of Russell's definite description operator (§2.4), which are now part of the Isabelle/HOL library.

On JinjaThreads' side, we animated the formalisation (see §3.1 for an overview) through code generation: Many of its inductive definitions, we had to refine for compilation or, if this was impossible, implement manually (§3.2 and §3.3). To obtain execution traces of JinjaThreads programs, we adapted the state representation and formalised two schedulers (§3.4). In §3.5, we explain how to add memoisation to avoid frequently recomputing common functions, e.g. lookup functions, without polluting the existing formalisation. Clearly, as the generated code naively interprets source code programs, we cannot expect it to be as efficient as an optimising Java virutal machine (JVM). Nevertheless, we evaluated the performance of the generated interpreter (§3.6). Simple optimisations that we describe there speed up the interpreter by three orders of magnitude. Hence, it is sufficiently efficient to handle Java programs of a few hundred lines of code.

We conclude our contributions by distilling our experience into a few guidelines on how to develop formalisations to be executable ones. Rather than imposing drastic changes on the formalisation, they pinpoint common pitfalls. §4 explains why and how to avoid them.

The interpreter and the full formalisation is available online in the Archive of Formal Proofs [17]. To make the vast supply of Java programs available for experimenting and testing with the semantics, we have written the (unverified) conversion tool Java2Jinja as a plug-in to the Eclipse IDE. It converts Java class declarations into JinjaThreads abstract syntax. The latest development version is available at `http://pp.info.uni-karlsruhe.de/git/Java2Jinja/`.

1.1 Related Work

Code generation (of *functional implementations*) from Isabelle/HOL is a well-established business. Marić [19] presents a formally verified implementation of a SAT solver. In the CeTA project, Thiemann and Sternagel [20] generate a self-contained executable termination checker for term rewriting systems. The Flyspeck project uses code generation to compute the set of tame graphs [4]. All these formalisations were developed with executability in mind. Complications in proofs to obtain an efficiently executable implementation were willingly taken and handling them are large contributions of these projects.

Code generation in Coq [13] has been used in various developments, notably the CompCert compiler [12] and the certificate checkers in the MOBIUS project

[3]. Like in Isabelle, functional specifications pose no intrinsic problems. Although code extraction is in principle possible for any Coq specification, mathematical theories can lead to "a nightmare in term of extracted code efficiency and readability" [13]. Hence, Coq's users, too, are facing the problem of how to extract (roughly) efficient code from specifications not aimed towards executability. ACL2 and PVS translate only functional implementations to Common Lisp.

In [5], we have reported on generating code from *non-functional specifications.* Recently, Nipkow applied code generation for inductive predicates to animate the semantics and various program analyses of an educational imperative language (personal communication). All these applications were tiny formalisations compared to JinjaThreads.

Some *formalisations of the JVM* in theorem provers are directly executable. The most complete is the M6 model of a JVM by Lui and Moore [14] in ACL2, which covers the CLDC specification. Farzan et al. [7] report on a JVM formalisation in Maude's rewriting logic. ACL2's and Maude's logics are directly executable, i.e., they force the user to write only executable formalisations. While JinjaThreads studies meta-language properties like type safety for a unified model of Java and Java bytecode, these JVM formalisations aim at verifying properties of individual programs. Atkey [1] presents an executable JVM model in Coq. He concentrates on encoding defensive type checks as dependent types, but does not provide any data on the efficiency.

1.2 Background: The Code Generator Framework and Refinement

Isabelle's code generator [9] turns a set of equational theorems into a functional program with the same equational rewrite system. As it builds on equational logic, the translation guarantees partial correctness by construction and the user may easily refine programs and data without affecting her formalisation globally. *Program refinement* can separate code generation issues from the rest of the formalisation. As any (executable) equational theorem suffices for code generation, the user may *locally* derive new (code) equations to use upon code generation. Hence, existing definitions and proofs remain unaffected, which has been crucial for JinjaThreads.

For *data refinement*, the user may replace constructors of a datatype by other constants and derive equations that pattern-match on these new (pseudo-)constructors. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type. Again, this is local as it affects only code generation, but not the logical properties of the refined type. Conversely, one cannot exploit the type's new structure inside the logic. Only type constructors can be refined; some special types (such as $'a \Rightarrow 'b$ *option* for maps) must first be wrapped in an (isomorphic) type of their own (e.g. $('a,'b)$ *mapping*).

Isabelle's standard library defines such special-purpose types for sets and maps with standard operations. Associative lists and red-black trees implement them via data refinement. FinFuns [18] are almost-everywhere constant functions; they provide an executable universal quantifier thanks to data refinement to associative lists. The Isabelle Collections Framework (ICF) [11] advocates dealing with refinement *in* the logic instead of hiding it in the code generator. Locales, i.e. Isabelle modules, specify the abstract operations, concrete implementations interpret them. This allows for executing truly underspecified functions.

2 Code Generation in Isabelle

In this section, we present our contributions that JinjaThreads has motivated, but that are generally applicable. Consequently, they have been integrated into Isabelle's main system and library. First, we present the code generator for inductive predicates and our improvements to it (§2.1). Then, we describe our approach to overcome the problematic situation with code generation and locales (§2.2). Finally, we sketch formalisations for enhanced implementations for the reflexive transitive closure (§2.3) and the definite description operator (§2.4), which are employed in JinjaThreads' type system, for example.

2.1 The Predicate Compiler

The *predicate compiler* [5] translates specifications of inductive predicates, i.e. the introduction rules, into executable equational theorems for Isabelle's code generator. The translation is based on the notion of *modes*. A mode partitions the arguments into input and output. For a given predicate, the predicate compiler infers the set of possible modes such that all terms are ground during execution. Lazy sequences handle the non-determinism of inductive predicates. By default, the equations implement a Prolog-style depth-first execution strategy. Since its initial description [5], we improved the predicate compiler in four aspects:

First, mode annotations restrict the generation of code equations to modes of interest. This is necessary because the set of modes is exponential in the number of arguments of a predicate. Therefore, the space and time consumption of the underlying mode inference algorithm grows exponentially in that number; for all applications prior to JinjaThreads, this has never posed a problem. In case of many arguments (up to 15 in JinjaThreads), the plain construction of this set of modes burns up any available hardware resource. To sidestep this limitation, modes can now be declared and hence they are not inferred, but only checked to be consistent.

Second, we also improved the compilation scheme: The previous one sequentially checked which of the introduction rules were applicable. Hence, the input values were repeatedly compared to the terms in the conclusion of each introduction rule by pattern matching. For large specifications, such as JinjaThreads' semantics (contains 88 rules), this naive compilation made execution virtually impossible due to the large number of rules. To obtain an efficient code expression, we modified the compilation scheme to partition the rules by patterns of the input values first and then only compose the matching rules – this resembles similar techniques in Prolog compilers, such as clause indexing and switch detection. We report on the performance improvements due to this modification in §3.6.

Third, the predicate compiler now offers non-intrusive program refinement, i.e., the user can declare alternative introduction rules. For an example, see §3.3.

Fourth, the predicate compiler was originally limited to the restricted syntactic form of introduction rules. We added some preprocessing that transforms definitions in predicate logic to a set of introduction rules. Type-safe method overriding (§3.2) gives an example.

2.2 Isabelle Locales and Code Generation

Locales [2] in Isabelle allow parametrised theory and proof development. In other words, locales allow to prove theorems abstractly, relative to a set of *fixed parameters and assumptions*. Interpretation of locales transfers theorems from their abstract context to other (concrete) contexts by instantiating the parameters and proving the assumptions. JinjaThreads uses locales to abstract over different memory consistency models (§3.3) and schedulers (§3.4), and to underspecify operations on abstract data structures.

As code generation requires equational theorems in the (foundational) theory context, theorems that reside in the context of a locale cannot serve as code equations directly, but must be transferred into the theory context. For example, consider a locale L with one parameter p, one assumption $A\ p$ and one definition $f = \ldots$ that depends on p. Let g be a function in the theory context for which $A\ (g\ z)$ holds for all z. We want to generate code for f where p is instantiated to $g\ z$.

The Isabelle code generator tutorial proposes *interpretation and definition*: One instantiates p by $g\ z$ and discharges the assumption with $A\ (g\ z)$, for arbitrary z. This yields the code equation $f\ (g\ z) = \ldots$ which is ill-formed because the left-hand side applies f to the non-constructor constant g. For code generation, one must manually define a new function f' by $f'\ z = f\ (g\ z)$ and derive $f'\ z = \ldots$ as code equation. This approach is unsatisfactory for two reasons: It requires to manually re-define all dependent locale definitions in the theory context (and for each interpretation), and the interpretation must be unconditional, i.e., $A\ (g\ z)$ must hold for *all* z. In JinjaThreads, the latter is often violated, e.g. $g\ z$ satisfies A only if z is well-formed.

To overcome these deficiencies, our new approach *splits the locale* L into two: L_0 and L_1. L_0 fixes the parameter p and defines f; L_1 inherits from L_0, assumes $A\ p$, and contains the proofs from L. Since L_0 makes no assumptions on p, the locale implementation exports the equation $f = \ldots$ in L_0 as an unconditional equation $L_0.f\ p = \ldots$ in the theory context which directly serves as code equation. For execution, we merely pass $g\ z$ to $L_0.f$. We use this scalable approach throughout JinjaThreads. Its drawback is that the existence of a model for f, as required for its definition, must not depend on L's assumptions; e.g. the termination argument of a general recursive function must not require L's assumptions. Many typical definitions (all in JinjaThreads) satisfy this restriction.

2.3 Tabling the Reflexive Transitive Closure

The reflexive transitive closure (RTC) is commonly used in formalisations, also in JinjaThreads' subtyping relation. Here, we present how a simple refinement implements a tabling depth-first execution of the RTC. By default, the predicate compiler uses the two introduction rules below for code generation.

$$\frac{}{\mathit{rtc}\ r\ x\ x} \qquad\qquad \frac{r\ x\ y \qquad \mathit{rtc}\ r\ y\ z}{\mathit{rtc}\ r\ x\ z}$$

Compiling them in a Prolog-style depth-first fashion leads to non-termination when the underlying relation r has reachable cycles. Hence, Berghofer implemented a tabled version of RTC that detects cycles and short-circuits the search in that case (cf. acknowledgements). The predicate $\mathit{rtc\text{-}tab}\ r\ xs\ x\ z$ expresses that z is reachable in r from x without visiting any node in xs:

$$\frac{}{\mathit{rtc\text{-}tab}\ r\ xs\ x\ x} \qquad\qquad \frac{x \notin \mathit{set}\ xs \qquad r\ x\ y \qquad \mathit{rtc\text{-}tab}\ r\ (x \cdot xs)\ y\ z}{\mathit{rtc\text{-}tab}\ r\ xs\ x\ z}$$

For execution, the terminating *rtc-tab* implements RTC via program refinement with the equality $\mathit{rtc}\ r = \mathit{rtc\text{-}tab}\ r\ []$.

2.4 An Executable Definite Description Operator

Russell's definite description operator ι and Hilbert's choice ε extract a deterministic function from a relational formulation. Like any underspecified function, they pose a challenge for code generation [8], because their axiomatisations are not unconditional equations. Hence, we can only execute them via program refinement, i.e., we must derive such an equation from the axiomatisation. This is only possible for inputs for which the specification fixes a unique return value, e.g. singleton sets for ι and ε, as any implementation returns a fully specified value. Now, we construct an executable implementation for ι using the predicate compiler. Our execution strategy is as follows: We enumerate all values satisfying the predicate. If there is exactly one such value, we return it; otherwise, we throw a exception. To enumerate values efficiently, we rely on the predicate compiler.

For technical reasons, it works in terms of the type $'a\ \mathit{pred}$ [5], which is isomorphic to $'a \Rightarrow \mathit{bool}$. The *Pred* constructor and the *eval* selector allow to convert between $'a\ \mathit{pred}$ and $'a \Rightarrow \mathit{bool}$. Then, the type of sequences $'a\ \mathit{seq}$ implements $'a\ \mathit{pred}$ via data refinement with the lazy constructor $\mathit{Seq} :: (\mathit{unit} \Rightarrow {'a}\ \mathit{seq}) \Rightarrow {'a}\ \mathit{pred}$. The type $'a\ \mathit{seq}$ has a richer structure with the constructors *Empty*, *Insert*, and *Join*. *Empty* and *Insert* are self-explanatory; $\mathit{Join}\ P\ xq$ represents the union of the enumeration P and the values in the sequence xq.

First, we lift ι to $'a\ \mathit{pred}$ by defining $\mathit{the}\ A\ \ = (\iota x.\ \mathit{eval}\ A\ x)$. Then, we define by (1) the operation $\mathit{singleton} :: (\mathit{unit} \Rightarrow {'a}) \Rightarrow {'a}\ \mathit{pred} \Rightarrow {'a}$ that returns for a singleton enumeration the contained element and a (lazy) *default* value otherwise. We prove (2) to implement *the* via *singleton*, which exploits reflexivity of HOL's equality for non-singleton enumerations.

$$\mathit{singleton}\ \mathit{default}\ A = (\mathit{if}\ \exists! x.\ \mathit{eval}\ A\ x\ \mathit{then}\ \iota x.\ \mathit{eval}\ A\ x\ \mathit{else}\ \mathit{default}\ ()) \tag{1}$$

$$\mathit{the}\ A = \mathit{singleton}\ (\lambda\_.\ \mathit{the}\ A)\ A \tag{2}$$

Having refined $'a\ \mathit{pred}$ to $'a\ \mathit{seq}$, we prove (3) as code equation for *singleton*:

$$
\begin{array}{l}
\mathit{singleton\ default}\ (\mathit{Seq}\ f) = (\mathit{case}\ f\ ()\ \mathit{of} \\
\qquad \mathit{Empty} \Rightarrow \mathit{throw\ default} \\
\quad \mid \mathit{Insert}\ x\ P\ \ \Rightarrow \mathit{if\ is\text{-}empty}\ P\ \mathit{then}\ x \\
\qquad\quad \mathit{else\ let}\ y = \mathit{singleton\ default}\ P\ \mathit{in\ if}\ x = y\ \mathit{then}\ x\ \mathit{else\ throw\ default} \\
\quad \mid \mathit{Join}\ P\ \mathit{xq}\ \ \Rightarrow \mathit{if\ is\text{-}empty}\ P\ \mathit{then\ the\text{-}only\ default\ xq} \\
\qquad\quad \mathit{else\ if\ null\ xq\ then\ singleton\ default}\ P \\
\qquad\qquad \mathit{else\ let}\ x = \mathit{singleton\ default}\ P;\ y = \mathit{the\text{-}only\ default\ xq\ in} \\
\qquad\qquad\quad \mathit{if}\ x = y\ \mathit{then}\ x\ \mathit{else\ throw\ default})
\end{array} \tag{3}
$$

The predicate *is-empty* (*null*) tests if the enumeration (the sequence) contains no element. The operation *the-only* is *singleton*'s analogon for $'a\ \mathit{seq}$ with a similar code equation. In HOL, *throw*, defined by $\mathit{throw}\ f = f\ ()$, just applies the *unit* value. The generated code for *throw* raises an exception without evaluating its argument, a *unit* closure. This ensures partial correctness for *singleton* and *the*, i.e., if the code terminates normally, the computed value is correct.

To execute definitions with Russell's ι operator, one proceeds as follows: Given a definition $c = (\iota x.\ P\ x)$, one runs the predicate compiler on P to obtain the function that enumerates x, i.e., the mode assigns the argument to be output. This yields an executable function $P\text{-}o$ with $P = \mathit{eval}\ P\text{-}o$. Unfolding definitions, one obtains the code equation $c = \mathit{the}\ P\text{-}o$.

Note that the test $x = y$ in (3) requires that equality on the predicate's elements is executable. If this is not the case (e.g. functions as elements), we provide an altered equation where *throw default* replaces $\mathit{if}\ x = y\ \mathit{then}\ x\ \mathit{else\ throw\ default}$ in (3). Then, the computation also fails when the enumeration is actually a singleton, but contains the same element multiple times.

3 JinjaThreads: Well-Formedness Checker and Interpreter

In this section, we first give an overview of JinjaThreads (§3.1). Then, we present how to obtain an executable well-formedness checker and interpreter for JinjaThreads, and what the pitfalls are (§3.2 to §3.4). We employ program and data refinement such that lookup functions are precomputed rather than recomputed whenever needed (§3.5). In §3.6, we evaluate the efficiency of the interpreter on a standard producer-consumer program.

3.1 Overview of JinjaThreads

Building on Jinja [10] by Klein and Nipkow, JinjaThreads models a substantial subset of multithreaded Java source and bytecode. Figure 1 shows the overall structure: The three major parts are the source and bytecode formalisations and a compiler between them. Source and bytecode share declarations of classes, fields and methods, the subtyping relation, and standard well-formedness constraints. The source code part defines the source code syntax, a single-threaded small-step semantics, and additional well-formedness constraints (such as a static

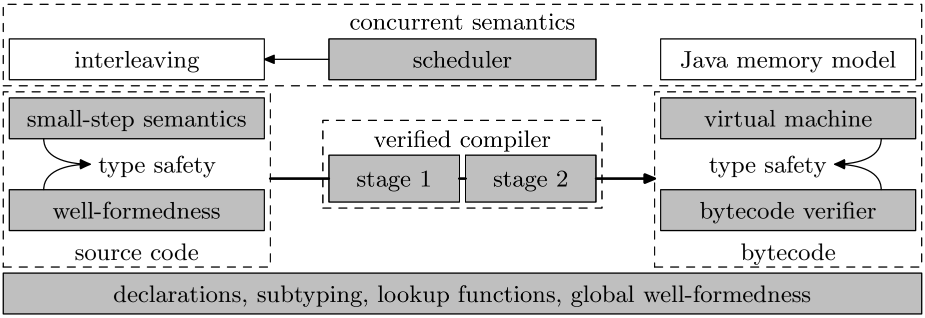


Fig. 1. Structure of JinjaThreads

type system and definite assignment). It contains a type safety proof via progress and preservation. The bytecode part formalises bytecode instructions, a virtual machine (VM) for individual threads, and a bytecode verifier. The type safety proof shows that verified bytecode cannot violate type checks in the defensive VM. For both parts, JinjaThreads defines two concurrent semantics: (i) interleaving semantics for the individual threads, which provides sequential consistency (SC) as memory consistency model (MCM) – schedulers allow to generate specific interleavings; (ii) the Java memory model (JMM) as an axiomatic specification of legal executions. Finally, the compiler translates source code into bytecode in two stages and is verified with respect to the concurrent semantics.

For all definitions in shaded boxes, we have generated code via Isabelle's code generator. We highlight the necessary steps using examples from well-formedness (§3.2), the small-step semantics (§3.3), and the scheduler (§3.4). The compiler's definition, a functional implementation, is directly executable. The bytecode verifier requires adaptations similar to well-formedness. For the VM, we had to manually transform its functional specification to use Isabelle's special-purpose type for sets (c.f. §1.2). The JMM is purely axiomatic, finding an operational model would be a complicated task that we have not attempted.

3.2 The Type System and Well-Formedness Conditions

A JinjaThreads program is given as a list of class declarations. Among others, *well-formedness* requires that its class hierarchy is acyclic with *Object* at the top, method overriding is type safe, and the program obeys the rules of the type system. Thus, a well-formedness checker must include a type checker. The type system relies on the subclass and subtype relation, least upper bounds (lub) w.r.t. the subtype relation, and lookup functions for fields and methods. To turn these into executable equations, we do the following:

The *subclass relation* $\preceq^*$ is the RTC of the direct subclass relation, which is defined inductively. As the standard execution mechanism for the RTC leads to non-termination in case of cyclic class hierarchies, we use the tabled RTC as described in §2.3. This ensures that querying $\preceq^*$ always terminates, i.e., we can reliably detect cyclic class hierarchies when checking well-formedness.

$$P, t \vdash \langle \mathit{null}.M(\mathit{map}\ \mathit{Val}\ \mathit{vs}), s\rangle \xrightarrow{\varepsilon} \langle \mathit{THROW}\ \mathit{NullPointer}, s\rangle \ \textsc{CallNull}$$

$$\frac{\mathit{is\text{-}Vals}\ \mathit{es}}{P, t \vdash \langle \mathit{null}.M(\mathit{es}), s\rangle \xrightarrow{\varepsilon} \langle \mathit{THROW}\ \mathit{NullPointer}, s\rangle}\ \textsc{CallNull2}$$

Fig. 2. Original and alternative introduction rule of the small-step semantics

The *subtype relation* $:\leq$, another inductive predicate, extends $\preceq^*$ to arrays and primitive types. Checking whether one type is a subtype of another is executable. For acyclic class hierarchies with *Object* at the top, non-primitive types form an upper semi-lattice w.r.t. $:\leq$, i.e. unique lubs exist for existing types. However, compiling the declarative definition of lub to an executable function with the predicate compiler fails, because it would require to enumerate all supertypes of a given type. Therefore, we provide a functional implementation, join, to compute lubs for acyclic class hierarchies with *Object* at the top. For cyclic ones, lubs need not be unique, so the functional implementation's behaviour is undefined.

Field and method lookup recurse over the class hierarchy. To avoid definitional problems in case of cyclic class hierarchies, JinjaThreads defines them relationally as inductive predicates and the lookup functions using the definite description operator ι. We refine them to use the executable operator *the* following §2.4.

The *type system* $E \vdash e :: T$ for source code statements is defined inductively, too. Even type checking requires type inference. Consider, e.g. the rule below for assignments to a local variable V whose type T is given by the environment E:

$$\frac{E\ V = \lfloor T \rfloor \qquad E \vdash e :: U \qquad U :\leq T \qquad V \neq \mathit{this}}{E \vdash V := e :: \mathit{Void}}$$

When the predicate compiler compiles $\_ \vdash \_ :: \_$, it must choose either to enumerate all subtypes of T and type-check e against each, or to infer e's type and check for $U :\leq T$. Note that in case $V := e$ is type-incorrect, the former approach might not terminate as e.g. *Object* has infinitely many subtypes. To force the predicate compiler to choose the latter, we disallow enumeration of subtypes via mode annotations.

For type inference, the rule for the conditional operator `_ ? _ : _` in Java requires to compute the lub of types of the second and third argument. As the declarative definition of the type system uses the declarative lub definition, type inference (and thus type checking) is not executable. For code generation, we therefore copy the definition for $\_ \vdash \_ :: \_$, replacing lub with the executable join function. Then, we prove that both versions agree on acyclic class hierarchies with *Object* at the top, but we cannot refine the declarative definition because equality only holds under acyclicity.

Overriding method M with parameter types Ts and return type T in class C with direct superclass D is *type-safe* if

$$\forall Ts'\ T'\ m.\ \vdash D\ \mathit{sees}\ M\colon Ts' \rightarrow T' = m \implies Ts'\ [:\leq]\ Ts \wedge T :\leq T'$$

where $\vdash D$ *sees* $M\colon Ts' \rightarrow T' = m$ denotes that D sees M with parameter types Ts', return type T' and body m. The predicate compiler preprocesses the condition to an inductive predicate and compiles it to an executable equation (cf. §2.1).

After these preparations, *well-formedness* no longer poses any difficulties for code generation. Note that all the setup relies on program refinement only, the existing formalisation remains untouched. Stating and proving alternative equations requires between 5 lines for $:\leq$ and 220 lines for the type system.

3.3 The Semantics

The small-step semantics is parametric in the MCM. Thus, we model shared memory abstractly in terms of read and write functions for values and type information as locale parameters, following the splitting principle from §2.2.

The small-step semantics for source code is another inductive predicate. The predicate compiler processes 84 of 88 introduction rules automatically. For the others, we must provide alternative introduction rules via program refinement. Fig. 2 shows the rule CallNull, which is representative for the four, for thread t invoking the method M with parameter values vs on the *null* pointer in the state s, which raises a *NullPointer* exception. Mapping the injection *Val* of values into expressions over the list of values vs expresses that all parameters have already evaluated to values. This rule violates the desired mode for executing the semantics because its execution would require pattern-matching against the term *map Val* vs. The remedy is to declare the alternative introduction rule CallNull2: We replace *map Val* vs by es and instead use the guard *is-Vals* es that predicates that all elements in es are of the form *Val* v for some v. To access vs in other parts of the rule (as is necessary in one of the others), we replace vs with *map the-Val* es where *the-Val* is the destructor for the constructor *Val*.

Mode annotations for executing the small-step semantics are crucial. The abstraction of the MCM in a locale adds 6 parameters to the small-step semantics in the theory context, which consequently allows a monstrous number of modes.

For code generation, we only use SC as MCM, because the JMM is axiomatic and thus not executable. SC models the shared heap as a function from addresses (natural numbers) to objects and arrays. Allocation must find a fresh address, i.e. one not in the heap's domain. Originally, this was defined via Hilbert's underspecified (and thus not executable) ε operator (4). For code generation, we had to change *new-Addr*'s specification to the least fresh address, replacing ε with *LEAST*. Then, we proved (5) and (6) to search for the least fresh address.

$$\mathit{new\text{-}Addr}\ h = \mathit{if}\ (\exists a.\ h\ a = \mathit{None})\ \mathit{then}\ \lfloor \varepsilon a.\ h\ a = \mathit{None} \rfloor\ \mathit{else}\ \mathit{None} \quad (4)$$

$$\mathit{new\text{-}Addr}\ h = \mathit{gen\text{-}new\text{-}Addr}\ h\ 0 \quad (5)$$

$$\mathit{gen\text{-}new\text{-}Addr}\ h\ n = \mathit{if}\ (h\ n = \mathit{None})\ \mathit{then}\ \lfloor n \rfloor\ \mathit{else}\ \mathit{gen\text{-}new\text{-}Addr}\ h\ (n{+}1) \quad (6)$$

3.4 The Scheduler

Executing the interleaving semantics poses three problems:

1. The multithreaded state consists of functions of type $\_ \Rightarrow \_$ *option* for locks, thread-local states and the monitor's wait sets. Neither quantifying over these maps' domains (e.g. to decide whether all threads have terminated) nor picking one of its elements (e.g. to remove an arbitrary thread from a wait set upon notification) are executable.
2. The state space of all possible interleavings is usually too large to be effectively enumerable. Therefore, one wants to pick one typical interleaving.
3. JinjaThreads programs that might not terminate should at least produce a prefix of the observable operations of such an infinite run.

To address the first, we previously [18] proposed to replace these maps with FinFuns, a generalisation of finite maps. Although quantification over the domain then becomes executable, it turned out that choosing an underspecified element remains unexecutable. We therefore only use them for lock management. For the pool of thread-local states and the wait set, we instead follow the ICF approach [11]. We replace the functions with abstract operations whose signatures and properties we specify in two locales. Picking an arbitrary element remains underspecified, but this is now explicit inside the logic, not HOL's metalogic. Before code generation, we instantiate the locales with concrete data structure implementations like red-black trees and thus resolve the underspecification.

As to the second problem, we do not use the predicate compiler, as it would produce a depth-first search that enumerates all possible interleavings. The first few interleavings would be such that one thread executes completely (or until it blocks), then the next thread executes completely, etc. Interesting interleavings would occur only very much later – or never at all, if one of the preceding ones did not terminate. Instead, we let a scheduler pick the next thread at each step.

Formally, a scheduler consists of two operations (that we specify abstractly in two locales again): The function *schedule* takes the scheduler's state and the multithreaded state, and returns either a thread together with its next transition and the updated scheduler state, or *None* to denote that the interleaving has finished or is deadlocked. The other function *wakeup* chooses from a monitor's wait set the thread to be notified. In terms of these two functions, we define a deterministic, executable step function that updates the multithreaded state just like the non-deterministic interleaving semantics does. To obtain a complete interleaving as a potentially infinite trace, we corecursively unfold this step function. Then, we formally prove that this in fact yields a possible interleaving.

We have instantiated this specification with two concrete schedulers: a round-robin scheduler and a random scheduler based on a pseudo-random number generator. The most intricate problem is how to obtain (as a function) the thread's step from the (relational) small-step semantics, once the scheduler has decided which thread to execute. Fortunately, the semantics under SC is deterministic, if we purge transitions whose preconditions are not met by the current state. Thus, we use the *the* operator again, but without equality checks (§2.4), as the result states contain functions (the heap) for which checking equality is not executable.

```
1 datatype 'm prog = Program 'm cdecl list
2 definition prog-impl-invar P' c s f m = (c = Mapping (class (Program P')) ∧ ...)
3 typedef 'm prog-impl = {(P', c, s, f, m) | prog-impl-invar P' c s f m}
  morphisms impl-of Abs-prog
4 definition ProgDecl = Program ∘ fst ∘ impl-of
5 code_datatype ProgDecl
6 lemma [code]: class (ProgDecl P) = lookup (fst (snd (impl-of P)))
7 definition tabulate P' = Abs-prog (P', tabulate-class P', tabulate-subcls P', ...)
8 lemma [code]: Program = ProgDecl ∘ tabulate
```

Fig. 3. Tabulation for lookup functions and the subclass relation

Corecursive traces also solve the third problem. We instruct the code generator to implement possibly infinite lists *lazily*. For Haskell, this is the default; for the other target languages, data and program refinement provide an easy setup.

Formalising the scheduler did not affect the rest of the formalisation. It required 2357 lines of definitions and proofs, 20% of which only declare locales.

3.5 Tabulation

An execution of a JinjaThreads (or, similarly, Java) program frequently checks type casts and performs method lookups. However, with the above setup, the semantics recomputes the subtype relation and lookup functions at every type cast and method call from scratch. Here, we show how to leverage program and data refinement to avoid such recomputations with only minimal changes to the formalisation itself. We precompute the subclass relation, field and method lookup (a standard technique for VMs) and store them in mappings (cf. §1.2). Fig. 3 sketches the necessary steps.

In JinjaThreads, a program declaration used to be a list of class declarations, i.e. of type *'m cdecl list*, abbreviated as *'m prog*. For data refinement (cf. §1.2), we turn the abbreviation into a type of its own, wrapping the old type (l. 1 in Fig. 3).

Next, we define the type *'m prog-impl* (l. 3). Apart from the original program declaration (as a list P'), its elements (P', c, s, f, m) consist of mappings from class names to (i) the class declaration (c), (ii) the set of its superclasses (s), and (iii) two mappings for field and method lookup with field and method names as keys (f and m). The invariant *prog-impl-invar* (l. 2) states that the mappings correctly tabulate the lookup functions and subclass relation. Then, we define (l. 4) and declare (l. 5) the new constructor *ProgDecl* :: *'m prog-impl* ⇒ *'m prog* for data refinement, which (in the logic) only extracts the program declaration.

For the lookup functions, the subclass relation, and the associated constants that the predicate compiler has introduced, we next prove code equations that implement them via lookup in the respective mapping – see l. 6 for class declaration lookup. This program refinement suffices to avoid recomputing lookup functions and the subclass relation during execution.

However, the generated code now expects the input program to come with the correctly precomputed mappings. Thus, we define *tabulate* (l. 7) and auxiliary functions that tabulate the lookup functions and subclass relation in these

Table 1. Timing (in seconds) for running the producer-consumer example on n objects for different adjustments to the interpreter; — denotes timeout after 1h

n	without adjustments	with indexing	almost strict	heap as red-black tree	with tabulation
10	229.9	1.9	.1	<.1	<.1
100	2,240.3	14.1	1.7	.7	.6
1,000	—	625.6	492.3	7.2	6.2
10,000	—	—	—	71.8	62.6

mappings for a given list P' of class declarations. Finally, we implement the former constructor *Program* (l. 8) in terms of *tabulate* and *ProgDecl*.

As most of JinjaThreads treats a program declaration opaquely, introducing *′m prog* as a type of its own was painless; we edited just 143 lines out of 70k, i.e. .2%. The remaining program and data refinement took about 600 lines.

3.6 Efficiency of the Interpreter

Although we cannot expect the generated interpreter to be as efficient as an optimising JVM, to see whether it is suited to run small programs, we have evaluated it on a standard producer-buffer-consumer example. The producer thread allocates n objects and enqueues them in the buffer, which can store 10 elements at the same time. Concurrently, the consumer thread dequeues n objects from the buffer. Table 1 lists the running times for different code generator setups. All tests ran on a Pentium DualCore E5300 2.6GHz with 2GB RAM using Poly/ML 5.4.1 and Ubuntu GNU/Linux 9.10.

With the adaptations from §3.2 to §3.4 only, the code is unbearably slow (column 1). For $n = 100$, interpreting the program takes 37 min, i.e. 2,240.3 s. As the main bottleneck, we identified the naive compilation scheme for the small-step semantics. By switching to the improved compilation scheme (column 2) in the predicate compiler (§2.1), we sped up the interpreter by two orders of magnitude. The definite descriptor *the* that extracts the result configuration from the enumerations, strictly evaluates all branches. Hence, explicit laziness in the generated code is unnecessary. If we remove the most obvious constructions due to laziness from the code equations that we compiled under the improved scheme, a program run with $n = 100$ takes only 1.7 s (column 3).

As n increases, another bottleneck shows up: memory allocation (cf. §3.3). Since the heap is modelled as a function and writes as function updates, i.e. closures, finding the next fresh address takes time quadratic in the number of previous allocations. Thus, interpreting the example program is quadratic in n although the program itself only requires linearly many steps. To speed up allocation and read access, we replaced the function by a red-black tree with addresses as keys. Combined with the other improvements, this already provides a decent interpreter (column 4): Run times grow linearly in n as expected.

Finally, we also added tabulation (cf. §3.5), where the mappings are for simplicity implemented as associative lists. Surprisingly, the speed-up (less than

15%) is modest. The reason might be the tiny class hierarchy of the example program for which lookups functions terminate quickly.

We also ran the tests with the code generated in Haskell (compiled with Glasgow Haskell Compiler 6.10.4) and OCaml (compiled to native code with OCaml 3.11.1). The Haskell code is about 60% slower than the ML and the OCaml code takes between 2 to 5 times as much time as ML. Still, the different adjustments to the interpreter affect the run times similarly to ML.

As JinjaThreads also has a verified compiler and a virtual machine, we also ran the virtual machine on the compiled code. The virtual machine is 6 to 7 times faster than the source code interpreter with red-black trees for the heap: Pushing 10,000 objects through the buffer takes 9.6 s with tabulation and 11.9 s without. Clearly, rewriting expressions in the small-step semantics is slower than pattern-matching on instructions. Still, our interpreter and VM are still far from a commercial VM: The Java HotSpot VM takes only 30 ms for 10,000 objects.

In [14], Lui and Moore test their JVM formalisation M6 in ACL2 on a simple parallel factorial algorithm. To compare our interpreter with theirs, we have converted the Java program to JinjaThreads with our Java2Jinja tool. For computing 10! with five threads in parallel, our source code semantics takes 26.7 s and the VM just 0.2 s. The M6 takes 6.2 s when run in the ACL2 interpreter, version 2.7 with GNU CLISP 2.42.

4 Guidelines for Executable Formalisations

From our experience with JinjaThreads, we have distilled the following guidelines to easily obtain executable formalisations in Isabelle.

Avoid Hilbert's ε operator! Hilbert's choice cannot express underspecification adequately as, in HOL's model, its interpretation is fully specified. Partial correctness of the code generator guarantees that all evaluations in the functional language hold in *every model*. Thus, one cannot replace it by any implementing function that chooses one suitable value consistently and fixes the underspecified function to *one concrete model*. Instead, use one of the following alternatives:

1. Change the definition to make the choice deterministic and implementable, e.g. always pick the least element.
2. Use locales for intra-logical underspecification and instantiate the choice operator to a concrete implementation by locale interpretation.
3. Switch to a relational description and prove the correctness for all values.

The first is least intrusive to the formalisation, but requires changes to the original specification. To execute the deterministic choice, one needs to run the predicate compiler on the choice property and use the executable definite descriptor for predicates (§2.4), or implement a suitable search algorithm via program refinement, as we did for memory allocation (§3.3).

The second is the most flexible, but also tedious as the locale does not automatically setup proof automation and lacks true polymorphism. We use this approach e.g. to specify schedulers §3.4. Care must be taken in combination with data refinement via the code generator, as the choice must not depend on the additional structure that the interpretation introduces.

The last option completely avoids underspecification, but relinquishes the functional implementation. For code generation, one should either (i) apply the predicate compiler to obtain code that computes *all* possible implementations for the specification, or (ii) provide a functional implementation and show correctness (cf. §3.4). For this, one must typically replace the involved types with others that have additional structure.

Structure locales wisely! Modular specifications, i.e. locales, and code generation do not (yet) go well together (cf. §2.2). To combine them, one best adheres to the following discipline: One locale *Sig* fixes the parameters' signatures and contains all definitions that depend on the parameters. Another locale *Spec* extends *Sig* and states the assumptions about the parameters; all proofs that depend on the properties go into *Spec*. For functions and inductive predicates of *Sig*, one feeds the equational theorems and introduction rules exported into the theory context to the code generator or predicate compiler, resp. To obtain the (correctness) theorems, instantiate *Spec* and prove the assumptions.

Annotate predicates with modes! Mode annotations for predicates instruct the predicate compiler to generate only modes of interest, not all modes that its mode analysis can infer. They provide three benefits. First, if the predicate has many parameters, analysing all modes can quickly become computationally intractable (cf. §3.3) – in this case, they are necessary. Second, they ease maintenance and debugging as they fail immediately after adjustments: If changes in the development disable a mode of interest, an error message indicates which clauses are to blame. Without annotations, the missing mode might remain undiscovered until much later, which then complicates correcting errors. Third, some not annotated, but inferable modes might lead to generation of slow or non-terminating functions. By disallowing them, the predicate compiler cannot accidentally pick one of them when it compiles a subsequent predicate.

5 Conclusion and Future Work

Originally, the JinjaThreads formalisation aimed to investigate semantics properties of concurrent Java; executability was of little concern throughout its development. At the start, subtleties in the formalisation inhibited executing the specifications. After we had substantially improved the code generation of inductive predicates and manually adapted and extended the formalisation, we obtained a Java interpreter with decent performance. We found solutions on how to marry code generation with locales and how to adequately handle underspecification and the definite description operator. From our experience, we extracted guidelines on how to develop future formalisations with executability in mind.

JinjaThreads' predecessor Jinja [10] has been developed eight years ago. Comparing the efforts and results to obtain executability, we note the following improvements: First, Jinja's code generator setup relied on manual and unsound translations, e.g. sets as raw lists and ad hoc implementations for Hilbert's ε operator. In contrast, we adapted the formalisation such that the unsound translations are no longer necessary. Instead, we use safe implementations for sets from Isabelle's library and model underspecification explicitly inside the logic. Second, the Jinja interpreter can loop infinitely when it executes ill-formed programs, but Jinja lacks a well-formedness checker. Employing our new implementations (cf. §2.3), JinjaThreads now offers a decision procedure for checking well-formedness. Third, the (now outdated) predicate compiler, which Jinja uses, generates code directly in the functional target language. Thus, interweaving purely functional and logical computations as, e.g. in the JinjaThreads scheduler would have been impossible within the logic, but required editing the generated code. Exploiting program and data refinement, we obtained a sound and executable definite description operator (§2.4) to link both worlds.

Thanks to these increased efforts, we reach a new level of confidence in the generated code, which would have been impossible with the tools eight years ago. Still, this extensive case study revealed some pressing issues for code generation:

To execute JinjaThreads' virtual machine specification we employ implementations for common set operations. The necessary refinement is conceptionally straightforward, but requires a tremendous effort if done manually. This step should be automated.

The lack of integration between locales and code generation requires all users to follow a rather strict discipline (cf. §2.2). A solution on Isabelle's side that integrates locales and code generation needs to be addressed in the future.

Acknowledgements. We thank F. Haftmann for invaluable help with the code generator, S. Berghofer for improving the RTC, J. Thedering and A. Zea for their work on Java2Jinja, and J. Blanchette, A. Popescu, M. Hecker, D. Lohner, and the anonymous reviewers for comments on earlier drafts. We acknowledge funding from DFG grants Sn11/10-1,2 and DFG doctorate program 1480 (PUMA).

References

1. Atkey, R.: CoqJVM: An Executable Specification of the Java Virtual Machine Using Dependent Types. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 18–32. Springer, Heidelberg (2008)
2. 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)
3. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS proof carrying code infrastructure. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 1–24. Springer, Heidelberg (2008)
4. Bauer, G., Nipkow, T.: Flyspeck I: Tame graphs. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (2006), http://afp.sourceforge.net/entries/Flyspeck-Tame.shtml, Formal proof development

5. 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)
6. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)
7. Farzan, A., Bevilacqua, V., Roşu, G.: Formal JVM code analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 147–150. Springer, Heidelberg (2004)
8. Haftmann, F.: Data refinement (raffinement) in Isabelle/HOL This is a draft of an envisaged publication still to be elaborated which, applying the usual rules of academic confidentiality, can be inspected at, `http://www4.in.tum.de/~haftmann/pdf/data_refinement_haftmann.pdf`
9. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
10. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Sys. 28, 619–695 (2006)
11. 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)
12. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning 43(4), 363–446 (2009)
13. Letouzey, P.: Extraction in Coq: An overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008)
14. Liu, H., Moore, J.S.: Executable JVM model for analytical reasoning: A study. In: IVME 2003, pp. 15–23. ACM, New York (2003)
15. Lochbihler, A.: Type safe nondeterminism – a formal semantics of Java threads. In: Workshop on Foundations of Object-Oriented Languages, FOOL 2008 (2008)
16. Lochbihler, A.: Verifying a compiler for Java threads. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 427–447. Springer, Heidelberg (2010)
17. Lochbihler, A.: Jinja with threads. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (2007), `http://afp.sourceforge.net/entries/JinjaThreads.shtml`, Formal proof development
18. Lochbihler, A.: Formalising FinFuns – generating code for functions as data from Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 310–326. Springer, Heidelberg (2009)
19. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
20. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009)

Formalization of Entropy Measures in HOL

Tarek Mhamdi, Osman Hasan, and Sofiène Tahar

ECE Department, Concordia University, Montreal, QC, Canada
{mhamdi,o_hasan,tahar}@ece.concordia.ca

Abstract. Information theory is widely used in a very broad class of scientific and engineering problems, including cryptography, neurobiology, quantum computing, plagiarism detection and other forms of data analysis. Despite the safety-critical nature of some of these applications, most of the information theoretic analysis is done using informal techniques and thus cannot be completely relied upon. To facilitate the formal reasoning about information theoretic aspects, this paper presents a rigorous higher-order logic formalization of some of the most widely used information theoretic principles. Building on fundamental formalizations of measure and Lebesgue integration theories for extended reals, we formalize the Radon-Nikodym derivative and prove some of its properties using the HOL theorem prover. This infrastructure is then used to formalize information theoretic fundamentals like Shannon entropy and relative entropy. We discuss potential applications of the proposed formalization for the analysis of data compression and security protocols.

1 Introduction

Information theory [19] was developed as a mathematical theory for communication by Claude E. Shannon to define the theoretical limits on the achievable performance of data compression and transmission rate of communication. The limits, being the entropy and the channel capacity, respectively, are given in terms of coding theorems for information sources and noisy channels. Information theory has since been used in analyzing the correctness and performance of a broad range of scientific and engineering systems, e.g., [18,5,12].

Traditionally, paper-and-pencil based analytical techniques have been used for information theoretic analysis but these methods do not scale very well to most real-world systems. Therefore, computer simulations are predominantly used for information theoretic analysis these days. However, due to its inherent nature, computer simulation can never ascertain 100% accuracy. This fact is extremely undesirable due to the ever increasing usage of information theoretic analysis in the design of safety and mission critical systems. Formal methods tend to overcome such inaccuracy limitations and therefore a higher-order-logic formalization of information theory has recently been proposed [3]. However, the underlying theories of this development have certain constraints and lack important properties of the quantities formalized, which are necessary for any information theoretic analysis. For example, the theories do not support infinite

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 233–248, 2011.

values for functions or integrals, which limits the scope of applications and most importantly prevents the proof of important and necessary theorems, like the Radon Nikodym theorem [7].

This paper is primarily focused towards overcoming these shortcomings as we attempt to raise the state-of-the-art in higher-order-logic theorem proving based information theoretic analysis technique from the existing level, where it is applicable only to isolated facets, to a level allowing formal analysis of contemporary engineering and scientific problems. In this regard, we propose to first develop a rigorous higher-order-logic formalization of measure, probability, Lebesgue integration theories over extended real numbers, which are real numbers extended with positive and negative infinity.

Using extended reals to define the measure theory allows us to work with regular finite non-negative measures, infinite measures as well as signed measures. Working with functions or random variables that can take infinite values, allows us to prove important limiting theorems that are not possible to prove when we do not consider infinite values. In fact, in that case, the limit of a sequence is undefined when the sequence is not convergent. However, in the extended reals case, a limit is always defined and can be infinite. Finally, working with infinite Lebesgue integrals allows us to prove various convergence theorems without requiring the sequences to be convergent.

Building on top of this framework, we formalize Shannon's entropy and the relative entropy, which are most widely used information theoretic principles, and verify their classical properties. In the definition of relative entropy, we need to define the Radon Nikodym derivative and prove its properties. The existence of this derivative for absolutely continuous measures is guaranteed by the so called Radon Nikodym theorem. The proof of this theorem was the main motivation to use the extended reals in the formalization.

All of the above mentioned formalization is done using the HOL theorem prover [8] and the paper provides the associated formalization and verification details. This infrastructure paves the path to the formal information theoretic analysis of many engineering systems and we highlight some of these potential applications of our work in this paper as well.

2 Related Work

Based on the work of Hurd [10] on measure theory, Coble [3] formalized the main concepts of Lebesgue integration and probability and used them in the formalization of information theory in HOL. Coble used this framework to verify anonymity properties of the dining cryptographers protocol. This formalization, however, does not include important convergence theorems and properties of the Lebesgue integral and measurable functions, limiting the scope of its applications. We provided a generalization of this work [13], based on Borel spaces, allowing us to verify those properties and theorems. Both formalizations, however, only consider finite-valued measures, functions and integrals. In this paper, we propose to define a new type for extended reals and use it to formalize measure, Lebesgue integration, probability and main concepts of information theory. Using extended reals in the formalization has many advantages. It allows us to define sigma-finite and other infinite measures as well as signed measures. Properties of the Lebesgue integral like the monotonicity can be proven even for non-integrable functions, but most importantly, it allows us to prove convergence theorems that are valid even for non convergent sequences. The latter was the main reason to define extended-real-valued integrals, to be able to prove the important Radon Nikodym theorem. This theorem, and consequently some of the properties of the Radon Nikodym derivative, could not be proven using the formalizations in [3,13]. The Radon Nikodym derivative is needed in the definition of the relative entropy. To the best our knowledge, this is the first higher-order-logic formalization of these information theoretic notions which also includes their properties.

A formalization of the positive extended reals in HOL was proposed by Hurd [11] and has been imported to the Isabelle theorem prover [16]. We propose a formalization that includes all real numbers as well as the positive infinity $+\infty$ and negative infinity $-\infty$. This has, obviously, the advantage of working with negative extended real numbers, for example for signed measures. A formalization of measure theory defined on the positive extended reals has been developed in Isabelle [9], based on the work of Coble [3]. This has been used to prove the Radon Nikodym theorem. The main difference with our work is the use of extended reals, which allows us to define signed measures as well as have the integral defined on the extended reals for arbitrary functions. Most importantly, in our work, we focus on defining the main concepts of information theory as well as prove their properties. We prove the properties of the Radon Nikodym derivative and use it to define and prove the properties of the relative entropy.

A formalization of the Lebesgue integral on the extended reals has been proposed in Mizar [20]. We provide a more general formalization that allowed us to formalize the Radon Nikodym derivative and prove its properties. To the best of our knowledge, neither the Radon Nikodym derivative and its properties nor the relative entropy have been formalized in Mizar.

3 Extended Real Numbers

The set of extended real numbers $\overline{\mathbb{R}}$ is the set of real numbers $\mathbb{R}$ extended with two additional elements, namely, the positive infinity $+\infty$ and negative infinity $-\infty$. $\overline{\mathbb{R}}$ is useful to describe various limiting behaviors in many mathematical fields. For instance, it is necessary to use the extended reals system to define the integration theory, otherwise the convergence theorems such as the monotone convergence and dominated convergence theorems would be less useful. Using the extended reals to define the measure theory makes it possible to define sigma finite measures and other infinite measures. With extended reals, the limit of a monotonic sequence is always defined, infinite when the sequence is divergent, but still defined and properties can be proven on it. The price to pay for these advantages is an increased level of difficulty in the analysis and the need to prove a large body of theorems on the extended reals and operators on them.

An extended real is either a normal real number, positive infinity or negative infinity. we use `Hol_datatype` to define the new type `extreal` as follows,

```
val _ = Hol_datatype‘extreal = NegInf | PosInf | Normal of real‘;
```

The arithmetic operations of $\mathbb{R}$ are extended to $\overline{\mathbb{R}}$ with partial functions. For example the addition is extended as follow.

$$\forall a.\ a \neq -\infty \Rightarrow a + (+\infty) = +\infty + a = +\infty$$
$$\forall a.\ a \neq +\infty \Rightarrow a + (-\infty) = -\infty + a = -\infty$$

This is formalized in higher-order logic as

```
val extreal_add_def = Define‘
   (extreal_add (Normal x) (Normal y) = (Normal (x + y))) ∧
   (extreal_add (Normal _) a = a) ∧
   (extreal_add b (Normal _) = b) ∧
   (extreal_add NegInf NegInf = NegInf) ∧
   (extreal_add PosInf PosInf = PosInf)‘
```

The function is left undefined when one of the operands is `PosInf` and the other is `NegInf`. Similarly, we extend the other arithmetic operators and prove their properties.

The set of extended real numbers is a totally ordered set such that for all $a \in \overline{\mathbb{R}}$, $-\infty \leq a \leq +\infty$. With this order, $\overline{\mathbb{R}}$ is a complete lattice where every subset has a supremum and an infimum. The supremum is formalized in HOL as:

```
val extreal_sup_def = Define
  ‘extreal_sup p =
   if ∀x. (∀y. p y ⇒ y ≤ x) ⇒ (x = PosInf) then PosInf
   else (if ∀x. p x ⇒ (x = NegInf) then NegInf
            else Normal (sup (λr. p (Normal r))))’;
```

In this definition, `sup` refers to the supremum over a set of real numbers. Next, we tackle the following theorem, which we will use in the Radon Nikodym theorem proof in Section 5

Theorem 1. *For any non-empty, upper bounded (by a finite number) set P of extended real numbers, there exists a monotonically increasing sequence of elements of P that converges to the supremum of P.*

For the case where the supremum is an element of the set, we simply consider the sequence $\forall n,\ x_p(n) = \sup P$. Otherwise, we prove that $x_p(n)$, defined below, is one such sequence.

$$x_p(0) = @r.\ r \in P \wedge (\sup P - 1) < r \text{ and}$$
$$x_p(n+1) = @r.\ r \in P \wedge \max(x_p(n), \sup P - \tfrac{1}{2^{n+1}}) < r < \sup P$$

where @ represents the Hilbert choice operator.
We then define the sum of extended real numbers over a finite set and prove its properties whenever the sum is defined. The obvious way to define the sum is the following

```
val SIGMA_DEF = new_definition("SIGMA_DEF",
  ‘‘SIGMA f s = ITSET (λe acc. f e + acc) s (0:extreal)’’)
```

However, using this definition, we are not able to prove the recursive form without requiring that all the elements we are adding are finite. In fact, to be able to prove the recursive form, we need to use the theorem

```
 ∀f e s b.
   (∀x y z. f x (f y z) = f y (f x z)) ∧ FINITE s ⇒
   (ITSET f (e INSERT s) b = f e (ITSET f (s DELETE e) b))
```

This requires that the addition is associative and commutative for all the elements considered, which is not the case unless we restrict our definition to finite values. This is, obviously, undesirable when working with extended real numbers. Instead, we propose the following definition for the sum.

```
val SIGMA_def = let open TotalDefn
 in tDefine "SIGMA"
    ‘SIGMA (f:’a -> extreal) (s: ’a -> bool) =
       if FINITE s then
          if s= then 0:extreal
          else f (CHOICE s) + SIGMA f (REST s)
       else ARB‘
  (WF_REL_TAC ‘measure (CARD o SND)‘ THEN
   METIS_TAC [CARD_PSUBSET, REST_PSUBSET])
 end;
```

We use `WF_REL_TAC` to initiate the termination proof of the definition with the measure function `measure (CARD o SND)`. From this definition, we prove the recursive form, which will be used in proving the main properties of the sum.

```
∀f s. FINITE s  ⇒
   ∀e. (∀x. x ∈ e INSERT s ⇒ f x ≠ NegInf) ∨
       (∀x. x ∈ e INSERT s ⇒ f x ≠ PosInf) ⇒
    (SIGMA f (e INSERT s) = f e + SIGMA f (s DELETE e))
```

Notice that we can have infinite values as long as the sum in defined. The properties that we proved include the linearity, monotonicity, and the summation over disjoint sets and products of sets.

Finally, we define the infinite sum of extended real numbers $\sum_{n\in\mathbb{N}} x_n$ using the `SIGMA` and `sup` operators and prove its properties.

```
val ext_suminf_def = Define
   ‘ext_suminf f = sup (IMAGE (λn. SIGMA f (count n)) UNIV)’
```

We provide an extensive formalization of the extended real numbers, which consists of more than 220 theorems written in around 3000 lines of code. It contains all the necessary tools to formalize most of the concepts that we need in measure, integration, probability and information theories. The proof script is available in [14] and can used in a variety of other applications as well. In the next sections, we present the formalization of these theories based on the extended real numbers.

4 Formalization of Measure, Integration and Probability

Using measure theory to formalize probability has the advantage of providing a mathematically rigorous treatment of probabilities and a unified framework for discrete and continuous probability measures. In this context, a probability measure is a measure function, an event is a measurable set and a random variable is a measurable function. The expectation of a random variable is its integral with respect to the probability measure. The Lebesgue integral is used because it provides a unique definition for discrete and continuous random variables, it handles a broader class of functions than the Reimann integral, and it exhibits a better behavior when it comes to interchanging limits and integrals. Most of the concepts of this section have already been formalized in HOL [13]. However, the formalization of this paper is based on the extended reals. In this context, the `limit` of a monotonically increasing sequence becomes the `supremum` and can be infinite. This allows us to verify various limiting properties and convergence theorems.

4.1 Measure Theory

By definition, measurable functions satisfy the condition that the inverse image of a measurable set is also measurable, which we formalize in higher-order logic as follows

```
 ⊢ ∀a b f. f ∈ measurable a b =
     sigma_algebra a ∧ sigma_algebra b ∧
     f ∈ (space a → space b) ∧
     ∀s. s ∈ subsets b ⇒ PREIMAGE f s ∩ space a ∈ subsets a
```

This definition applies to functions defined on arbitrary spaces. We are interested in real-valued measurable functions and hence the Borel sigma algebra on the set of extended real numbers is used. Working with the Borel sigma algebra makes the set of measurable functions a vector space. It also allows us to formally verify various properties of the measurable functions necessary for the formalization of the Lebesgue integral and its properties in HOL.

We define the Borel sigma algebra on $\overline{\mathbb{R}}$, which we call *Borel*, as the smallest sigma algebra generated by the open rays

```
val Borel_def = Define
   ‘Borel = sigma (UNIV:extreal->bool)
                  (IMAGE (λa. {x:extreal | x < a}) UNIV)’
```

where `sigma` is defined as

```
sigma sp st = (sp, ⋂ s | st ⊆ s ∧ sigma_algebra (sp,s))
```

We also prove that the Borel sigma algebra on the extended reals is the smallest sigma algebra generated by any of the following classes of intervals: $[c, +\infty]$, $(c, +\infty]$, $[-\infty, c]$, (c, d), $[c, d)$, $(c, d]$, $[c, d]$, where $c, d \in \mathbb{R}$. Using the above result, we prove that to check the measurability of extended-real-valued function, it is sufficient to check that the inverse image of the open ray is measurable. The same result is valid for the other classes of intervals.

Theorem 2. *Let $(X, \mathcal{A})$ be a measurable space. A function $f : X \to \overline{\mathbb{R}}$ is measurable with respect to $(\mathcal{A}, \mathcal{B}(\overline{\mathbb{R}}))$ iff $\forall c \in \overline{\mathbb{R}},\ f^{-1}([-\infty, c[) \in \mathcal{A}$*

We prove in HOL various properties of the extended-real-valued measurable functions.

- Every constant real function on a space X is measurable.
- The indicator function on a set A is measurable iff A is measurable.
- Let f and g be measurable functions and $c \in \mathbb{R}$, then the following functions are also measurable: $cf, |f|, f^n, f + g, fg$ and $max(f, g)$.
- If (f_n) is a monotonically increasing sequence of real-valued measurable functions such that $\forall\, x,\ f(x) = \sup_{n \in}\ \ f_n(x)$, then f is a measurable function.

4.2 Lebesgue Integral

The Lebesgue integral is defined using a special class of functions called positive simple functions. They are measurable functions taking finitely many values. In other words, a positive simple function g is represented by the triple (s, a, x) as a finite linear combination of indicator functions of measurable sets (a_i) that form a partition of the space X.

$$\forall t \in X,\ g(t) = \sum_{i \in s} x_i I_{a_i}(t) \quad c_i \geq 0 \tag{1}$$

We also add the condition that positive simple functions take finite values, i.e., $\forall i \in s.\ x_i < \infty$. Their Lebesgue integral can however be infinite.

The Lebesgue integral is first defined for positive simple functions then extended to non-negative functions and finally to arbitrary functions. Let $(X, \mathcal{A}, \mu)$ be a measure space. The integral of the positive simple function g with respect to the measure μ is given by

$$\int_X g \, d\mu = \sum_{i \in s} x_i \mu(a_i) \tag{2}$$

This is formalized in HOL as

```
val pos_simple_fn_integral_def = Define
    ‘pos_simple_fn_integral m s a x =
            SIGMA (λi. x i * measure m (a i)) s’
```

While the choice of $((x_i), (a_i), s)$ to represent g is not unique, we prove that the integral as defined above is independent of that choice. We also prove important properties of the Lebesgue integral of positive simple functions such as the linearity and monotonicity. The Lebesgue integral of non-negative measurable functions is given by

$$\int_X f \, d\mu = \sup\{\int_X g \, d\mu \mid g \leq f \text{ and } g \text{ positive simple function}\} \tag{3}$$

Its formalization in HOL is the following

```
val pos_fn_integral_def = Define
    ‘pos_fn_integral m f =
            sup {r | ∃g. r ∈ psfis m g ∧ ∀x. g x ≤ f x}’
```

where `psfis m g` is used to represent the Lebesgue integral of the positive simple function g. Finally, the integral for arbitrary measurable functions is given by

$$\int_X f \, d\mu = \int_X f^+ \, d\mu - \int_X f^- \, d\mu \tag{4}$$

where f^+ and f^- are the non-negative measurable functions defined by $f^+(x) = \max(f(x), 0)$ and $f^-(x) = \max(-f(x), 0)$.

```
val fn_integral_def = Define
    ‘fn_integral m f = pos_fn_integral m (fn_plus f) -
                       pos_fn_integral m (fn_minus f)’
```

As defined above, the Lebesgue integral can be undefined when the integrals of both f^+ and f^- are infinite. This requires that in most properties of the Lebesgue integral, we assume that the functions are integrable, as defined next.

Definition 1. *Let $(X, \mathcal{A}, \mu)$ be a measure space, a measurable function f is integrable iff $\int_X f^+ \, d\mu < \infty$ and $\int_X f^- \, d\mu < \infty$*

Lebesgue Monotone Convergence. The monotone convergence is arguably the most important theorem of the Lebesgue integration theory and it plays a major role in the proof of the Radon Nikodym theorem [1] and the properties of the integral. We present in the sequel a proof of the theorem in HOL.

Theorem 3. *Let (f_n) be a monotonically increasing sequence of non-negative measurable functions such that $\forall\, x,\ f(x) = \sup_{n\in}\ f_n(x)$, then*

$$\int_X f \, d\mu = \sup_{n\in} \int_X f_n \, d\mu$$

```
 ⊢ ∀m f fi. measure_space m ∧ ∀i x. 0 ≤ fi i x ∧
    ∀i. fi i ∈ measurable (m_space m, measurable_sets m) Borel ∧
    ∀x. mono_increasing (λi. fi i x) ∧
    ∀x. x ∈ m_space m ⇒ f x = sup (IMAGE (λi. fi i x) UNIV) ⇒
        pos_fn_integral m f =
            sup (IMAGE (λi. pos_fn_integral m (fi i)) UNIV)
```

We prove the Lebesgue monotone convergence theorem by using the properties of the supremum and by proving the lemma stating that if f is the supremum of a monotonically increasing sequence of non-negative measurable functions f_n and g is a positive simple function such that $g \leq f$, then the integral of g satisfies

$$\int_X g\, d\mu \leq \sup_{n\in} \int_X f_n\, d\mu$$

Lebesgue Integral Properties. Most properties of the Lebesgue integral cannot be proved directly from the definition of the integral. We prove instead that any measurable function is the limit of a sequence of positive simple functions. The properties of the Lebesgue integral are then derived from the properties on the positive simple functions.

Theorem 4. *For any non-negative measurable function f there exists a monotonically increasing sequence of positive simple functions (f_n) such that $\forall\, x,\ f(x) = \sup_{n\in}\ f_n(x)$. Besides*

$$\int_X f\, d\mu = \sup_{n\in} \int_X f_n\, d\mu$$

The above theorem is formalized in HOL as

```
⊢  ∀m f. measure_space m ∧  ∀x. 0 ≤ f x ∧
     f ∈ measurable (m_space m,measurable_sets m) Borel ⇒
     ∃fi ri. ∀x. mono_increasing (λi. fi i x) ∧
     ∀x. x ∈ m_space m ⇒ sup (IMAGE (ı. fi i x) UNIV) = f x ∧
     ∀i. ri i ∈ psfis m (fi i) ∧
     pos_fn_integral m f =
                sup (IMAGE (λi. pos_fn_integral m (fi i)) UNIV)
```

We prove this theorem by showing that the sequence (f_n), defined below, satisfies the conditions of the theorem and use the Lebesgue monotone convergence theorem to conclude that $\int_X f\, d\mu = \sup_{n\in}\ \int_X f_n\, d\mu$.

$$f_n(x) = \sum_{k=0}^{4^n-1} \frac{k}{2^n} I_{\{x|\frac{k}{2^n} \leq f(x) < \frac{k+1}{2^n}\}} + 2^n I_{\{x|2^n \leq f(x)\}}$$

For arbitrary integrable functions, Theorem 4 is applied to f^+ and f^- and results in a well-defined integral, given by

$$\int_X f\, d\mu = \sup_{n\in} \int_X f_n^+\, d\mu - \sup_{n\in} \int_X f_n^-\, d\mu$$

Using Theorem 4, we extend the properties of the Lebesgue integral for positive simple functions to arbitrary integrable functions. The main properties we proved are the monotonicity and linearity of the Lebesgue integral.

4.3 Probability Theory

We formalize the Kolmogorov axiomatic definition of probability using measure theory by defining the sample space Ω, the set F of events which are subsets of Ω and the probability measure p. A probability measure is a measure function and an event is a measurable set. (Ω, F, p) is a probability space iff it is a measure space and $p(\Omega) = 1$. A random variable is by definition a measurable function.

```
val random_variable_def = Define
   ‘random_variable X p s = prob_space p ∧
                            X ∈ measurable (p_space p, events p) s’
```

The properties we proved in the previous section for measurable functions are obviously valid for random variables.

Theorem 5. *If X and Y are random variables and $c \in \mathbb{R}$ then the following functions are also random variables: $cX, |X|, X^n, X+Y, XY$ and $max(X,Y)$.*

The probability mass function (PMF) of a random variable X is defined as the function p_X assigning to a set A the probability of the event $\{X \in A\}$. We also formalize the joint probability mass function of two random variables and of a sequence of random variables.

```
val pmf_def = Define
        ‘pmf p X = (λA. prob p (PREIMAGE X A ∩ p_space p))’
```

Finally we use the formalization of the Lebesgue integral to define the expectation of a random variable and its variance. The expectation of a random value X is defined as the integral of X with respect to the probability measure, $E[X] = \int_{\Omega} X \, dp$.

```
val expectation_def = Define ‘expectation = fn_integral’
```

The properties of the expectation are derived from the properties of the integral. The variance of a random variable is defined as $E[|X - E[X]|^2]$. We also prove the properties of the variance in HOL.

5 Measures of Entropy in HOL

In this section, we make use of the formalization of measure, Lebesgue integral and probability theory to formalize fundamental quantities of information theory, namely the Shannon entropy and the relative entropy. We prove some of their properties and present some of their applications. In the definition of relative entropy, we need to define the Radon Nikodym derivative [7] and prove its properties. The existence of this derivative for absolutely continuous measures is guaranteed by the so called Radon Nikodym theorem [7]. The proof of this theorem was the main motivation to use the extended reals in the formalization.

5.1 Shannon Entropy

The Shannon entropy [4] is a measure of the uncertainty associated with a random variable. It is restricted to discrete random variables and its extension to continuous random variables, known as the differential entropy, does not have some of the desired properties. In fact, the differential entropy can be negative and is not invariant under change of variables.

Definition 2. *(Shannon Entropy) The entropy H of a discrete random variable X with alphabet $\mathcal{X}$ and probability mass function p is defined by*

$$H(X) = -\sum_{x \in \mathcal{X}} p(x)log(p(x))$$

We provide, by contrast, a formalization that is based on the expectation and is valid for both discrete and continuous cases. We prove, later, the equivalence between the two definitions.

$$H(X) = E[-log(p(X))]$$

We propose the following formalization of the entropy in higher-order logic.

```
⊢ entropy b p X =  expectation q (λx. - logr b (pmf p X {x}))
```

where, p is the probability space and b is the basis of the logarithm and q is the probability space with respect to which the expectation is defined and is given by

```
⊢  q = (IMAGE X (p_space p), POW (IMAGE X (p_space p)), pmf p X)
```

We then prove the equivalence between the two definitions of entropy, i.e. the expectation based definition and the sum based definition, for the case of a discrete random variable.

```
⊢  entropy b p X = -SIGMA (λx. pmf p X x * logr b (pmf p X {x}))
                                        (IMAGE X (p_space p))
```

We prove the Asymptotic Equipartition Property (AEP) [4] which is the information theoretic analog of the Weak Law of Large Numbers (WLLN) [15]. It states that for a stochastic source X, if its time series $X_1, X_2, \ldots$ is a sequence of independent identically distributed (*iid*) random variables with entropy $H(X)$, then $-\frac{1}{n}log(p(X_1, \ldots, X_n))$ converges in probability to $H(X)$. We prove the AEP by first proving the Chebyshev's inequality and use it to prove the WLLN.

Theorem 6. *(AEP): if $X_1, X_2, \ldots$ are iid then*

$$-\frac{1}{n}log(p(X_1, \ldots, X_n)) \longrightarrow H(X) \textit{ in probability}$$

A consequence of the AEP is the fact that the set of observed sequences, $(x_1, \ldots, x_n)$, for which the joint probabilities $p(x_1, x_2, \ldots, x_n)$ are close to $2^{-nH(X)}$, has a total probability equal to 1. This set is called the *typical set* and such sequences are called the *typical sequences.* In other words, out of all possible sequences, only a small number of sequences will actually be observed and those sequences are nearly equally probable. The AEP guarantees that any property that holds for the typical sequences is true with high probability and thus determines the average behavior of a large sample.

Definition 3. *(Typical Set) The typical set A_ϵ^n with respect to $p(x)$ is the set of sequences $(x_1, \ldots, x_n)$ satisfying*

$$2^{-n(H(X)+\epsilon)} \leq p(x_1, \ldots, x_n) \leq 2^{-n(H(X)-\epsilon)}$$

We use the AEP to prove that the typical set has a total probability equal to 1 and that the total number of typical sequences is upper bounded by $2^{n(H(X)+\epsilon)}$.

5.2 Relative Entropy

The relative entropy [4] or Kullback Leibler divergence $D(\mu||\nu)$ is a measure of the distance between two distributions μ and ν. It is defined as

$$D(\mu||\nu) = -\int_X log \frac{d\nu}{d\mu}\, d\mu$$

where $\frac{d\nu}{d\mu}$ is the Radon Nikodym derivative of ν with respect to μ. This derivative is a non-negative measurable function that, when it exists, satisfies for any measurable set.

$$\int_A \frac{d\nu}{d\mu}\, d\mu = \nu(A)$$

The Radon Nikodym derivative is formalized in HOL as

```
val RN_deriv_def = Define
  ‘RN_deriv m v =
    @f. f IN measurable (m_space m, measurable_sets m) Borel ∧
    (∀a. a ∈ measurable_sets m ⇒
     (fn_integral m (λx. f x * indicator_fn a x) = measure v a))’
```

The relative entropy is then formalized as

```
val KL_divergence_def = Define
  ‘KL_divergence b m v =
     - fn_integral m (λx. logr b ((RN_deriv m v) x))’
```

The existence of the Radon Nikodym derivative is guaranteed for absolutely continuous measures by the Radon Nikodym theorem. A measure ν is absolutely continuous with respect to the measure μ iff for every measurable set A, $\mu(A) = 0$ implies that $\nu(A) = 0$. Next, we state and prove the Radon Nikodym theorem (RNT) for finite measures. The theorem can be easily generalized to sigma finite measures.

Theorem 7. *(RNT) If ν is absolutely continuous with respect to μ, then there exists a non-negative $\mu-$integrable function f such that for any measurable sets,*

$$\int_A f\, d\mu = \nu(A)$$

The Radon Nikodym theorem is formalized in HOL as follows,

```
⊢ ∀m v. measure_space m ∧ measure_space v ∧
  (m_space v = m_space m) ∧
  (measurable_sets v = measurable_sets m) ∧
  (measure_absolutely_continuous m v) ∧
  (measure v (m_space v) ≠ PosInf) ∧
  (measure m (m_space m) ≠ PosInf) ⇒
  (∃f. f ∈ measurable (m_space m,measurable_sets m) Borel ∧
  (∀A. A ∈ measurable_sets m ⇒
   (pos_fn_integral m (λx. f x * indicator_fn A x) = measure v A)))
```

To prove the theorem, we prove the following lemma, which we propose as a generalization of Theorem 1. To the best of our knowledge, this lemma has not been referred to in textbooks and we find that it is a useful result that can be used in other proofs.

Lemma 1. *If P is a non-empty set of extended-real valued functions closed under the max operator, g is monotone over P and $g(P)$ is upper bounded, then there exists a monotonically increasing sequence $f(n)$ of functions, elements of P, such that*

$$\sup_{n\in\mathbb{N}} g(f(n)) = \sup_{f\in P} g(f)$$

Proving the Radon Nikodym theorem consists in defining the set F of non-negative measurable functions such that for any measurable set A, $\int_A f\, d\mu \leq \nu(A)$. Then we prove that this set is non-empty, upper bounded by the finite measure of the space and is closed under the max operator. Next, using the monotonicity of the integral and the lemma above, we prove the existence of a monotonically increasing sequence $f(n)$ of functions in F such that

$$\sup_{n\in\mathbb{N}} \int_X f_n\, d\mu = \sup_{f\in F} \int_X f\, d\mu$$

Finally, we prove that the function g, defined below, satisfies the conditions of the theorem.

$$\forall x.\ g(x) = \sup_{n\in\mathbb{N}} f_n(x)$$

The main reason we used the extended reals in our formalization was the inability to prove the Radon Nikodym theorem without considering infinite values. In fact, in our proof, we use the Lebesgue monotone convergence to prove that

$$\int_X g\, d\mu = \sup_{n\in\mathbb{N}} \int_X f_n\, d\mu$$

However, the Lebesgue monotone convergence in [13] which does not support the extended reals, requires the sequence f_n to be convergent, which is not necessarily the case here and cannot be added as an assumption because the sequence f_n is generated inside the proof. The Lebesgue monotone convergence theorem with the extended reals is valid even for sequences that are not convergent since it uses the `sup` operator instead of the limit `lim`.

Next, we prove the following properties of the Radon Nikodym derivative.

- The Radon Nikodym derivative of ν with respect to μ is unique, μ almost-everywhere, i.e., unique up to a null set with respect to μ.
- If ν_1 and ν_2 are absolutely continuous with respect to μ, then $\frac{d(\nu_1+\nu_2)}{d\mu} = \frac{d\nu_1}{d\mu} + \frac{d\nu_2}{d\mu}$, μ almost-everywhere.
- If ν is absolutely continuous with respect to μ and $c \geq 0$, then $\frac{d(c*\nu)}{d\mu} = c * \frac{d\nu}{d\mu}$, μ almost-everywhere.

For finite spaces, we prove the following two results for the Radon Nikodym derivative and the relative entropy.

$$\forall x \in X, \mu\{x\} \neq 0 \Rightarrow \frac{d\nu}{d\mu}(x) = \frac{\nu\{x\}}{\mu\{x\}}$$

$$\forall x \in X, \nu\{x\} \neq 0 \Rightarrow D(\mu||\nu) = \sum_{x \in X} \mu\{x\} \log \frac{\mu\{x\}}{\nu\{x\}}$$

Finally, the relative entropy between the joint distribution $p(x,y)$ of two random variables X and Y and the product of their marginal distributions $p(x)$ and $p(y)$ is equal to the mutual information $I(X,Y)$.

$$I(X,Y) = D(p(x,y)||p(x)p(y)) = \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

5.3 Applications

The developed formalization of entropy measures can be used in a number of engineering applications. For instance, the formally verified AEP and the typical set, formalized in Section 5.1, can be directly applied in the proof of the Shannon source coding theorem which establishes the fundamental limit of data compression. It states that it is possible to compress the data at a rate that is arbitrarily close to the Shannon entropy without significant loss of information. In other words, n iid random variables with entropy $H(X)$ can be expressed on the average by $nH(X)$ bits without significant risk of information loss, as n tends to infinity.

One way to prove the above theorem is to propose an encoding scheme that is based on the typical set. The average codeword length for all sequences is close to the average codeword length considering only the typical sequences, because, asymptotically, the total probability of the typical set is equal to 1. From the upper bound on the number of typical sequences, we deduce that the average number of bits needed to encode the typical sequences can be made arbitrarily close to $nH(X)$.

Quantitative theories of information flow are gaining a lot of attention in a variety of contexts, such as secure information flow, anonymity protocols, and side-channel analysis. Various measures are being proposed to quantify the flow of information. Serjantov [18] and Diaz et al. [6] independently proposed to use entropy to define the quality of anonymity and to compare different anonymity systems. In this technique, the attacker assigns probabilities to the users after observing the system and does not make use of any apriori information he/she might have. The attacker simply assumes a uniform distribution among the users before observation.

Deng [5] proposed the relative entropy as a measure of the amount of information revealed to the attacker after observing the outcomes of the protocol, together with the apriori information. We can use our formalization of the relative entropy developed in Section 5.2 to apply this technique to verify the anonymity properties of the Dining Cryptographers [2] and Crowds [17] protocols.

6 Conclusions

In this paper, we have presented a formalization in HOL of measure, Lebesgue integration and probability theories defined on the extended reals. We used this infrastructure, to formalize main concepts of information theory, namely the Shannon entropy and relative entropy. The formalization based on the extended reals enables us to verify important properties and convergence theorems as well as prove the important Radon Nikodym theorem. The latter allows us to prove the properties of the Radon Nikodym derivative, used in the definition of the relative entropy.

The verification of properties of the Shannon entropy and relative entropy makes it possible to perform information theoretic analysis on a wide range of applications. Using our formalization, we proved the Asymptotic Equipartition Property in HOL, which is used to define and verify the notion of typical sets. This, in turn, is the basis to prove the Shannon source coding theorem, providing the fundamental limits of data compression. The relative entropy is an important measure of divergence between probability distributions. It is used to define other concepts of information theory, but it is also used in several other applications like the anonymity application in [5].

Our future work include applying the technique in [5] to verify the anonymity properties of the Dining Cryptographers and Crowds protocols within the sound core of a theorem prover. We also plan to work out the details of the applications outlined in Section 5.3.

The HOL code for the formalization presented in this paper is available in [14]. It required more than 11000 lines of code and contains around 500 theorems. Most of this formalization is very generic and thus can be utilized to formalize more advanced mathematics or formally reason about a more wide range of engineering applications.

References

1. Bogachev, V.I.: Measure Theory. Springer, Heidelberg (2006)
2. Chaum, D.: The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. Journal of Cryptology 1(1), 65–75 (1988)
3. Coble, A.R.: Anonymity, Information, and Machine-Assisted Proof. PhD thesis, University of Cambridge (2010)
4. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley Interscience, Hoboken (1991)
5. Deng, Y., Pang, J., Wu, P.: Measuring Anonymity with Relative Entropy. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2006. LNCS, vol. 4691, pp. 65–79. Springer, Heidelberg (2007)
6. Díaz, C., Seys, S., Claessens, J., Preneel, B.: Towards Measuring Anonymity. In: Dingledine, R., Syverson, P.F. (eds.) PET 2002. LNCS, vol. 2482, pp. 54–68. Springer, Heidelberg (2003)
7. Goldberg, R.R.: Methods of Real Analysis. Wiley, Chichester (1976)
8. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
9. Hölzl, J.: Mechanized Measure, Probability, and Information Theory. Technical University Munich, Germany (2010), http://puma.in.tum.de/p-wiki/images/d/d6/szentendre_hoelzl_probability.pdf
10. Hurd, J.: Formal Verifcation of Probabilistic Algorithms. PhD thesis, University of Cambridge (2002)
11. Hurd, J., McIver, A., Morgan, C.: Probabilistic Guarded Commands Mechanized in HOL. Electronic Notes in Theoretical Computer Science 112, 95–111 (2005)
12. Malacaria, P.: Assessing Security Threats of Looping Constructs. SIGPLAN Not. 42(1), 225–235 (2007)
13. 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)
14. Mhamdi, T., Hasan, O., Tahar, S.: Formalization of Measure and Lebesgue Integration over Extended Reals in HOL. Technical Report, ECE Dept., Concordia University (February 2011), http://hvg.ece.concordia.ca/Publications/TECH_REP/MLX_TR11/
15. Papoulis, A.: Probability, Random Variables, and Stochastic Processes. Mc-Graw Hill, New York (1984)
16. C. Paulson, L.: Isabelle: a Generic Theorem Prover. Springer, Heidelberg (1994)
17. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for Web Transactions. ACM Transactions on Information and System Security 1(1), 66–92 (1998)
18. Serjantov, A., Danezis, G.: Towards an Information Theoretic Metric for Anonymity. In: Dingledine, R., Syverson, P.F. (eds.) PET 2002. LNCS, vol. 2482, pp. 41–53. Springer, Heidelberg (2003)
19. Shannon, C.E.: A Mathematical Theory of Communication. The Bell System Technical Journal 27(3), 379–423 (1948)
20. Shidama, Y., Endou, N., Kawamoto, P.N.: On the formalization of lebesgue integrals. Studies in Logic, Grammar and Rhetoric 10(23), 167–177 (2007)

On the Generation of Positivstellensatz Witnesses in Degenerate Cases*

David Monniaux[1] and Pierre Corbineau[2]

[1] CNRS, VERIMAG
[2] Université Joseph Fourier, VERIMAG

Abstract. One can reduce the problem of proving that a polynomial is nonnegative, or more generally of proving that a system of polynomial inequalities has no solutions, to finding polynomials that are sums of squares of polynomials and satisfy some linear equality (*Positivstellensatz*). This produces a *witness* for the desired property, from which it is reasonably easy to obtain a formal proof of the property suitable for a proof assistant such as Coq.

The problem of finding a witness reduces to a feasibility problem in semidefinite programming, for which there exist numerical solvers. Unfortunately, this problem is in general not strictly feasible, meaning the solution can be a convex set with empty interior, in which case the numerical optimization method fails. Previously published methods thus assumed strict feasibility; we propose a workaround for this difficulty.

We implemented our method and illustrate its use with examples, including extractions of proofs to Coq.

1 Introduction

Consider the following problem: given a conjunction of polynomial equalities, and (wide and strict) polynomial inequalities, with integer or rational coefficients, decide whether this conjunction is satisfiable over $\mathbb{R}$; that is, whether one can assign real values to the variables so that the conjunction holds. A particular case is showing that a given polynomial is nonnegative.

The decision problem for real polynomial inequalities can be reduced to *quantifier elimination*: given a formula F, whose atomic formulas are polynomial (in)equalities, containing quantifiers, provide another, equivalent, formula F', whose atomic formulas are still polynomial (in)equalities, containing no quantifier. An algorithm for quantifier elimination over the theory of *real closed fields* (roughly speaking, $(\mathbb{R}, 0, 1, +, \times, \leq)$ was first proposed by Tarski [27,30], but this algorithm had non-elementary complexity and thus was impractical. Later, the *cylindrical algebraic decomposition* (CAD) algorithm was proposed [7], with a doubly exponential complexity, but despite improvements [8] CAD is still slow in practice and there are few implementations available.

Quantifier elimination is not the only decision method. Basu et al. [2, Theorem 3] proposed a satisfiability testing algorithm with complexity $s^{k+1}d^{O(k)}$, where s is

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

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 249–264, 2011.

the number of distinct polynomials appearing in the formula, d is their maximal degree, and k is the number of variables. We know of no implementation of that algorithm. Tiwari [31] proposed an algorithm based on rewriting systems that is supposed to answer in reasonable time when a conjunction of polynomial inequalities has no solution.

Many of the algebraic algorithms are complex, which leads to complex implementations. This poses a methodology problem: can one trust their results? The use of computer programs for proving lemmas used in mathematical theorems was criticized in the case of Thomas Hales' proof of the Kepler conjecture. Similarly, the use of complex decision procedures (as in the proof assistant PVS[1]) or program analyzers (as, for instance, Astrée[2]) in order to prove the correctness of critical computer programs is criticized on grounds that these verification systems could themselves contain bugs.

One could formally prove correct the implementation of the decision procedure using a proof assistant such as Coq [12, 20]; but this is likely to be long and difficult. An alternative is to arrange for the procedure to provide a *witness* of its result. The answer of the procedure is correct if the witness is correct, and correctness of the witness can be checked by a much simpler procedure, which can be proved correct much more easily.

Unsatisfiability witnesses for systems of complex equalities or *linear* rational inequalities are already used within DPLL(T) satisfiability modulo theory decision procedures [17, ch. 11] [10]. It is therefore tempting to seek unsatisfiability witnesses for systems of polynomial inequalities.

In recent years, it was suggested [21] to use numerical *semidefinite programming* to look for proof witnesses whose existence is guaranteed by a *Positivstellensatz* [16,26,29]. The original problem of proving that a system of polynomial inequalities has no solution is reduced to: given polynomials P_i and R, derived from those in the original inequalities, find polynomials Q_i that are sums of squares such that $\sum_i P_i Q_i = R$. Assuming some bounds on the degrees of Q_i, this problem is in turn reduced to a *semidefinite programming* pure feasibility problem [6, 32], a form of convex optimization. The polynomials Q_i then form a witness, from which a machine-checkable formal proof, suitable for tools such as Coq [12] or Isabelle [11], may be constructed.

Unfortunately, this method suffers from a caveat: it applies only under a *strict feasibility* condition [22]: a certain convex geometrical object should not be degenerate, that is, it should have nonempty interior. Unfortunately it is very easy to obtain problems where this condition is not true. Equivalently, the method of rationalization of certificates [13] has a limiting requirement that the rationalized moment matrix remains positive semidefinite.

In this article, we explain how to work around the degeneracy problem: we propose a method to look for rational solutions to a general SDP feasibility problem. We have implemented our method and applied it to some examples

[1] http://pvs.csl.sri.com/

[2] http://www.astree.ens.fr/

from the literature on positive polynomials, and to examples that previously published techniques failed to process.

2 Witnesses

For many interesting theories, it is trivial to check that a given valuation of the variables satisfies a quantifier-free formula. A satisfiability decision procedure will in this case tend to seek a *satisfiability witness* and provide it to the user when giving a positive answer.

In contrast, if the answer is that the problem is not satisfiable, the user has to trust the output of the satisfiability testing algorithm, the informal meaning of which is "I looked carefully everywhere and did not find a solution." In some cases, it is possible to provide *unsatisfiability witnesses*: solutions to some form of dual or auxiliary problem that show that the original problem had no solution.

2.1 Nonnegativity Witnesses

To prove that a polynomial P is nonnegative, one simple method is to express it as a sum of squares of polynomials. One good point is that the degree of the polynomials involved in this sum of squares can be bounded, and even that the choice of possible monomials is constrained by the Newton polytope of P, as seen in §3.

Yet, there exist nonnegative polynomials that cannot be expressed as sums of squares, for instance this example due to Motzkin [24]:

$$M = x_1^6 + x_2^4 x_3^2 + x_2^2 x_3^4 - 3x_1^2 x_2^2 x_3^2 \tag{1}$$

However, Artin's answer to Hilbert's seventeenth problem is that any nonnegative polynomial can be expressed as a sum of squares of rational functions.[3]

It follows that such a polynomial can always be expressed as the quotient Q_2/Q_1 of two sums of squares of polynomials, which forms the nonnegativity witness, and can be obtained by solving $P.Q_1 - Q_2 = 0$ for $Q_1 \neq 0$ (this result is also a corollary of Th. 1).

2.2 Unsatisfiability Witnesses for Polynomial Inequalities

For the sake of simplicity, we shall restrict ourselves to wide inequalities (the extension to mixed wide/strict inequalities is possible). Let us first remark that the problem of testing whether a set of wide inequalities with coefficients in a subfield K of the real numbers is satisfiable over the real numbers is equivalent to the problem of testing whether a set of *equalities* with coefficients K is satisfiable over the real numbers: for each inequality $P(x_1, \ldots, x_m) \geq 0$, replace it by

[3] There exists a theoretical exact algorithm for computing such a decomposition for homogeneous polynomials of at most 3 variables [15]; we know of no implementation of it and no result about its practical usability.

$P(x_1, \dots, x_m) - \mu^2 = 0$, where μ is a new variable. Strict inequalities can also be simulated as follows: $P_i(x_1, \dots, x_m) \neq 0$ is replaced by $P_i(x_1, \dots, x_m).\mu = 1$ where μ is a new variable. One therefore does not gain theoretical simplicity by restricting oneself to equalities.

Stengle [29] proved two theorems regarding the solution sets of systems of polynomial equalities and inequalities over the reals (or, more generally, over real closed fields): a *Nullstellensatz* and a *Positivstellensatz*; a similar result was proved by Krivine [16]. Without going into overly complex notations, let us state consequences of these theorems.

Let K be an ordered field (such as $\mathbb{Q}$) and K' be a real closed field containing K (such as the real field $\mathbb{R}$), and let $\mathbf{X}$ be a list of variables $X_1, \dots, X_n$. A^{*2} denotes the squares of elements of A. The *multiplicative monoid* generated by A is the set of products of zero of more elements from A. The *ideal* generated by A is the set of sums of products of the form PQ where $Q \in K[\mathbf{X}]$ and $P \in A$. The *positive cone* generated by A is the set of sums of products of the form $p.P.Q^2$ where $p \in K$, $p > 0$, P is in the multiplicative monoid generated by A, and $Q \in K[\mathbf{X}]$. Remark that we can restrict P to be in the set of products of elements of A where no element is taken twice, with no loss of generality.

The result [9, 18, 19] of interest to us is:

Theorem 1. *Let $F_>$, $F_\geq$, $F_=$, $F_{\neq}$ be sets of polynomials in $K[\mathbf{X}]$, to which we impose respective sign conditions > 0, ≥ 0, $= 0$, $\neq 0$. The resulting system is unsatisfiable over ${K'}^n$ if and only if there exist an equality in $K[\mathbf{X}]$ of the type $S + P + Z = 0$, with S in the multiplicative monoid generated by $F_> \cup F_{\neq}^{*2}$, P belongs to the positive cone generated by $F_\geq \cup F_>$, and Z belongs to the ideal generated by $F_=$.*

(S, P, Z) then constitute a *witness* of the unsatisfiability of the system.[4]

For a simple example, consider the following system, which obviously has no solution:

$$\begin{cases} -2 + y^2 \geq 0 \\ 1 - y^4 \geq 0 \end{cases} \tag{2}$$

A *Positivstellensatz* witness is $y^2(-2 + y^2) + 1(1 - y^4) + 2y^2 + 1 = 0$. Another is $\left(\frac{2}{3} + \frac{y^2}{3}\right)(-2 + y^2) + \frac{1}{3}(1 - y^4) + 1 = 0$.

Consider the conjunction C: $P_1 \geq 0 \wedge \dots \wedge P_n \geq 0$ where $P_i \in \mathbb{Q}[X_1, \dots, X_m]$. Consider the set Π of products of the form $\prod_i P_i^{w_i}$ for $\mathbf{w} \in \{0, 1\}^n$ — that is, the set of all products of the P_i where each P_i appears at most once. Obviously, if one can exhibit nonnegative functions Q_j such that $\sum_{T_j \in \Pi} Q_j T_j + 1 = 0$, then C does not have solutions. Theorem 1 guarantees that if C has no solutions, then such functions Q_j exist as sum of squares of polynomials (we simply apply the theorem with $F_> = F_{\neq} = \emptyset$ and thus $S = \{1\}$). We have again reduced our problem to the following problem: given polynomials T_j and R, find sums-of-squares polynomials Q_j such that $\sum_j Q_j T_j = R$. Because of the high cost of

[4] Another result, due to Schmüdgen [26], gives simpler witnesses for $P_1 \geq 0 \wedge \dots \wedge P_n \geq 0 \Rightarrow C$ in the case where $P_1 \geq 0 \wedge \dots \wedge P_n \geq 0$ defines a compact set.

enumerating all products of the form $\prod_i P_i^{w_i}$, we have first looked for witnesses of the form $\sum_{T_j \in S} Q_j P_j + 1 = 0$.

3 Solving the Sums-of-Squares Problem

In §2.1 and §2.2, we have reduced our problems to: given polynomials $(P_j)_{1 \leq j \leq n}$ and R in $\mathbb{Q}[X_1, \dots, X_m]$, find polynomials that are sums of squares Q_j such that

$$\sum_j P_j Q_j = R \tag{3}$$

We wish to output the Q_j as $Q_j = \sum_{i=1}^{n_j} \alpha_{ji} L_{ji}^2$ where $\alpha_{ji} \in \mathbb{Q}^+$ and L_{ji} are polynomials over $\mathbb{Q}$. We now show how to solve this equation.

3.1 Reduction to Semidefinite Programming

Lemma 1. *Let $P \in K[X, Y, \dots]$ be a sum of squares of polynomials $\sum_i P_i^2$. Let $M = \{m_1, \dots, m_{|M|}\}$ be a set such that each P_i can be written as a linear combination of elements of M (M can be for instance the set of monomials in the P_i). Then there exists a $|M| \times |M|$ symmetric positive semidefinite matrix Q with coefficients in K such that $P(X, Y, \dots) = [m_1, \dots, m_{|M|}] Q [m_1, \dots, m_{|M|}]^T$, noting v^T the transpose of v.*

Assume that we know the M_j, but we do not know the matrices $\hat{Q}_j$. The equality $\sum_j P_j (M_j \hat{Q}_j (M_j)^T) = R$ directly translates into a system (S) of affine linear equalities over the coefficients of the $\hat{Q}_j$: $\sum_j (M_j \hat{Q}_j (M_j)^T) P_j - R$ is the zero polynomial, so its coefficients, which are affine linear combinations of the coefficients of the $\hat{Q}_j$ matrices, should be zero; each of these combinations thus yields an affine linear equation. The additional requirement is that the $\hat{Q}_j$ are positive semidefinite.

One can equivalently express the problem by grouping these matrices into a block diagonal matrix $\hat{Q}$ and express the system (S) of affine linear equations over the coefficients of $\hat{Q}$. By exact rational linear arithmetic, we can obtain a system of generators for the solution set of (S): $\hat{Q} \in -F_0 + \mathrm{vect}(F_1, \dots, F_m)$. The problem is then to find a positive semidefinite matrix within this *search space*; that is, find $\alpha_1, \dots, \alpha_m$ such that $-F_0 + \sum_i \alpha_i F_i \succeq 0$. This is the problem of *semidefinite programming*: finding a positive semidefinite matrix within an affine linear variety of symmetric matrices, optionally optimizing a linear form [6, 32].

For instance, the second unsatisfiability witness we gave for constraint system 2 is defined, using monomials $\{1, y\}$, 1 and $\{1, y\}$, by:

$$\left(\begin{array}{cc|c|cc} \frac{2}{3} & 0 & & & \\ 0 & \frac{1}{3} & & & \\ \hline & & \frac{1}{3} & & \\ \hline & & & 0 & 0 \\ & & & 0 & 0 \end{array}\right)$$

It looks like finding a solution to Equ. 3 just amounts to a SDP problem. There are, however, several problems to this approach:

1. For the general *Positivstellensatz* witness problem, the set of polynomials to consider is exponential in the number of inequalities.
2. Except for the simple problem of proving that a given polynomial is a sum of squares, we do not know the degree of the Q_j in advance, so we cannot[5] choose finite sets of monomials M_j. The dimension of the vector space for Q_j grows quadratically in $|M_j|$.
3. Some SDP algorithms can fail to converge if the problem is not *strictly feasible* — that is, the solution set has empty interior, or, equivalently, is not full dimensional (that is, it is included within a strict subspace of the search space).
4. SDP algorithms are implemented in floating-point. If the solution space is not full dimensional, they tend to provide solutions $\hat{Q}$ that are "almost" positive semidefinite (all eigenvalues greater than $-\epsilon$ for some small positive ϵ), but not positive semidefinite.

Regarding problem 1, bounds on degrees only matter for the *completeness* of the refutation method: we are guaranteed to find the certificate if we look in a large enough space. They are not needed for *soundness*: if we find a correct certificate by looking in a portion of the huge search space, then that certificate is correct regardless. This means that we can limit the choice of monomials in M_j and hope for the best.

Regarding the second and third problems : what is needed is a way to reduce the dimension of the search space, ideally up to the point that the solution set is full dimensional. As recalled by [22], in a sum-of-square decomposition of a polynomial P, only monomials $x_1^{\alpha_1} \dots x_n^{\alpha_n}$ such that $2(\alpha_1, \dots, \alpha_n)$ lies within the Newton polytope[6] of P can appear [23, Th. 1]. This helps reduce the dimension if P is known in advance (as in a sum-of-squares decomposition to prove positivity) but does not help for more general equations.

Kaltofen et al. [14] suggest solving the SDP problem numerically and looking for rows with very small values, which indicate useless monomials that can be safely removed from the basis; in other words, they detect "approximate kernel vectors" from the canonical basis. Our method is somehow a generalization of theirs: we detect kernel vectors whether or not they are from the canonical basis.

In the next section, we shall investigate the fourth problem: how to deal with solution sets with empty interior.

3.2 How to Deal with Degenerate Cases

In the preceding section, we have shown how to reduce the problem of finding unsatisfiability witnesses to a SDP feasibility problem, but pointed out one

[5] There exist non-elementary bounds on the degree of the monomials needed [19]. In the case of Schmüdgen's result on compact sets, there are better bounds [26].

[6] The Newton polytope of a polynomial P, or in Reznick's terminology, its *cage*, is the convex hull of the vertices $(\alpha_1, \dots, \alpha_n)$ such that $x_1^{\alpha_1} \dots x_n^{\alpha_n}$ is a monomial of P.

crucial difficulty: the possible degeneracy of the solution set. In this section, we explain more about this difficulty and how to work around it.

Let $\mathcal{K}$ be the cone of positive semidefinite matrices. We denote by $M \succeq 0$ a positive semidefinite matrix M, by $M \succ 0$ a positive definite matrix M. The vector $\mathbf{y}$ is decomposed into its coordinates y_i. $\tilde{x}$ denotes a floating-point value close to an ideal real value x.

We consider a SDP feasibility problem: given a family of symmetric matrices $F_0, F_i, \ldots, F_m$, find $(y_i)_{1 \leq i \leq m}$ such that

$$F(\mathbf{y}) = -F_0 + \sum_{i=1}^{m} y_i F_i \succeq 0. \tag{4}$$

The F_i have rational coefficients, and we suppose that there is at least one rational solution for $\mathbf{y}$ such that $F(\mathbf{y}) \succeq 0$. The problem is how to find such a solution.

If nonempty, the solution set $S \subseteq \mathbb{R}^m$ for the $\mathbf{y}$, also known as the *spectrahedron*, is semialgebraic, convex and closed; its boundary consists in $\mathbf{y}$ defining singular positive semidefinite matrices, its interior are positive definite matrices. We say that the problem is strictly feasible if the solution set has nonempty interior. Equivalently, this means that the convex S has dimension m.

Interior point methods used for semidefinite feasibility, when the solution set has nonempty interior, tend to find a solution $\tilde{\mathbf{y}}$ in the interior away from the boundary. Mathematically speaking, if $\tilde{\mathbf{y}}$ is a numerical solution in the interior of the solution set, then there is $\epsilon > 0$ such that for any $\mathbf{y}$ such that $\|\mathbf{y} - \tilde{\mathbf{y}}\| \leq \epsilon$, $\mathbf{y}$ is also a solution. Choose a very close rational approximation $\mathbf{y}$ of $\tilde{\mathbf{y}}$, then unless we are unlucky (the problem is almost degenerate and all any suitable ϵ is extremely small), then $\mathbf{y}$ is also in the interior of S. Thus, $F(\mathbf{y})$ is a solution of problem 4.

This is why earlier works on sums-of-square methods [22] have proposed finding rational solutions only when the SDP problem is strictly feasible. In this article, we explain how to do away with the strict feasibility clause.

Some problems are not strictly feasible. Geometrically, this means that the linear affine space $\{-F_0 + \sum_{i=1}^{m} y_i F_i \mid (y_1, \ldots, y_m) \in \mathbb{R}^m\}$ is tangent to the semidefinite positive cone $\mathcal{K}$. Alternatively, this means that the solution set is included in a strict linear affine subspace of $\mathbb{R}^m$. Intuitively, this means that we are searching for the solution in "too large a space"; for instance, if $m = 2$ and $\mathbf{y}$ lies in a plane, this happens if the solution set is a point or a segment of a line. In this case, some SDP algorithms may fail to converge if the problem is not strictly feasible, and those that converge, in general, will find a point slightly outside the solution set. The main contribution of this article is a workaround for this problem.

3.3 Simplified Algorithm

We shall thus now suppose the problem has empty interior.

The following result is crucial but easily proved:

Lemma 2. *Let E be a linear affine subspace of the $n \times n$ symmetric matrices such that $E \cap \mathcal{K} \neq \emptyset$. F in the relative interior I of $E \cap \mathcal{K}$. Then it follows:*

1. *For all $F' \in E \cap \mathcal{K}$, $\ker F \subseteq \ker F'$.*
2. *The least affine space containing $E \cap \mathcal{K}$ is $H = \{M \in E \mid \ker M \supseteq \ker F\}$.*

Suppose we have found a numerical solution $\tilde{\mathbf{y}}$, but it is nearly singular — meaning that it has some negative eigenvalues extremely close to zero. This means there is $\mathbf{v} \neq \mathbf{0}$ such that $|\mathbf{v}.F(\tilde{\mathbf{y}})| \leq \epsilon \|\mathbf{v}\|$. Suppose that $\tilde{\mathbf{y}}$ is very close to a rational solution $\mathbf{y}$ and, that $\mathbf{v}.F(\mathbf{y}) = 0$, and also that $\mathbf{y}$ is in the relative interior of S — that is, the interior of that set relative to the least linear affine space containing S. Then, by lemma 2, all solutions $F(\mathbf{y}')$ also satisfy $\mathbf{v}.F(\mathbf{y}') = 0$. Remark that the same lemma implies that either there is no rational solution in the relative interior, or that rational solutions are dense in S.

How can finding such a $\mathbf{v}$ help us? Obviously, if $\mathbf{v} \in \bigcap_{i=0}^{m} \ker F_i$, its discovery does not provide any more information than already present in the linear affine system $-F_0 + \text{Vect } (F_1, \ldots, F_m)$. We thus need to look for a vector outside that intersection of kernels; then, knowing such a vector will enable us to reduce the dimension of the search space from m to $m' < m$.

Thus, we look for such a vector in the orthogonal complement of $\bigcap_{i=0}^{m} \ker F_i$, which is the vector space generated by the rows of the symmetric matrices $F_0, \ldots, F_m$. We therefore compute a full rank matrix B whose rows span the exact same space; this can be achieved by echelonizing a matrix obtained by stacking $F_0, \ldots, F_m$. Then, $\mathbf{v} = \mathbf{w}B$ for some vector $\mathbf{w}$. We thus look for $\mathbf{w}$ such that $G(\mathbf{y}).\mathbf{w} = 0$, with $G(\mathbf{y}) = BF(\mathbf{y})B^T$.

The question is how to find such a $\mathbf{w}$ with rational or, equivalently, integer coefficients. Another issue is that this vector should be “reasonable” — it should not involve extremely large coefficients, which would basically amplify the floating-point inaccuracies.

We can reformulate the problem as: find $\mathbf{w} \in \mathbb{Z}^m \setminus \{0\}$ such that both $\mathbf{w}$ and $G(\tilde{\mathbf{y}}).\mathbf{w}$ are “small”, two constraints which can be combined into a single objective to be minimized $\alpha^2 \|G(\tilde{\mathbf{y}}).\mathbf{w}\|_2^2 + \|\mathbf{w}\|_2^2$, where $\alpha > 0$ is a coefficient for tuning how much we penalize large values of $\|G(\tilde{\mathbf{y}}).\mathbf{w}\|_2$ in comparison to large values of $\|\mathbf{w}\|_2$. If α is large enough, the difference between $\alpha G(\tilde{\mathbf{y}})$ and its integer rounding M is small. We currently choose $\alpha = \alpha_0 / \|G(\tilde{\mathbf{y}})\|$, with $\|M\|$ the Frobenius norm of M (the Euclidean norm for $n \times n$ matrices being considered as vectors in $\mathbb{R}^{n^2}$), and $\alpha_0 = 10^{15}$.

We therefore try searching for a small (with respect to the Euclidean norm) nonzero vector that is an integer linear combination of the $l_i = (0, \ldots, 1, \ldots, 0, m_i)$ where m_i is the i-th row of M and the 1 is at the i-th position. Note that, because of the diagonal of ones, the l_i form a free family.

This problem is known as finding a short vector in an integer lattice, and can be solved by the Lenstra-Lenstra-Lovász (LLL) algorithm. This algorithm outputs a free family of vectors s_i such that s_1 is very short. Other vectors in the family may also be very short.

Once we have such a small vector $\mathbf{w}$, using exact rational linear algebra, we can compute $F_0', \ldots, F_{m'}'$ such that

$$\left\{ -F_0' + \sum_{i=1}^{m'} y_i' F_i' \mid (y_1, \ldots, y_{m'}) \in \mathbb{R}^{m'} \right\} = \left\{ -F_0 + \sum_{i=1}^{m} y_i F_i \mid (y_1, \ldots, y_m) \in \mathbb{R}^{m} \right\} \cap \{F \mid F.\mathbf{v} = 0\} \quad (5)$$

The resulting system has lower search space dimension $m' < m$, yet the same solution set dimension. By iterating the method, we eventually reach a search space dimension equal to the dimension of the solution set.

If we find no solution F_0', then it means that the original problem had no solution (the Positivstellensatz problem has no solution, or the monomial bases were too small), or that a bad vector $\mathbf{v}$ was chosen due to lack of numerical precision. This is the only bad possible outcome of our algorithm: it may fail to find a solution that actually exists; in our experience, this happens only on larger problems (search space of dimension 3000 and more), where the result is sensitive to numerical roundoff. In contrast, our algorithm may never provide a wrong result, since it checks for correctness in a final phase.

3.4 More Efficient Algorithm

In lieu of performing numerical SDP solving on $F = -F_0 + \sum y_i F_i \succeq 0$, we can perform it in lower dimension on $-(BF_0B^T) + \sum y_i(BF_iB^T) \succeq 0$. Recall that the rows of B span the orthogonal complement of $\bigcap_{i=0}^{m} \ker F_i$, which is necessarily included in $\ker F$; we are therefore just leaving out dimensions that always provide null eigenvalues.

The reduction of the sums-of-squares problem (Eq. 3) provides matrices with a fixed block structure, one block for each P_j: for a given problem all matrices $F_0, F_1, \ldots, F_m$ are block diagonal with respect to that structure. We therefore perform the test for positive semidefiniteness of the proposed $F(\mathbf{y})$ solution block-wise (see Sec. 3.6 for algorithms). For the blocks not found to be positive semidefinite, the corresponding blocks of the matrices B and $F(\tilde{\mathbf{y}})$ are computed, and LLL is performed.

As described so far, only a single $\mathbf{v}$ kernel vector would be supplied by LLL for each block not found to be positive semidefinite. In practice, this tends to lead to too many iterations of the main loop: the dimension of the search space does not decrease quickly enough. We instead always take the first vector $\mathbf{v}^{(1)}$ of the LLL-reduced basis, then accept following vectors $\mathbf{v}^{(i)}$ if $\|\mathbf{v}^{(i)}\|_1 \leq \beta.\|\mathbf{v}^{(1)}\|_1$ and $\|G(\tilde{\mathbf{y}}).\mathbf{v}^{(i)}\|_2 \leq \gamma.\|G(\tilde{\mathbf{y}}).\mathbf{v}^{(1)}\|_2$. For practical uses, we took $\beta = \gamma = 10$.

When looking for the next iteration $\tilde{\mathbf{y}'}$, we use the $\tilde{\mathbf{y}}$ from the previous iteration as a hint: instead of starting the SDP search from an arbitrary point, we start it near the solution found by the previous iteration. We perform least-square minimization so that $-F_0' + \sum_{i=1}^{m'} y_i' F_i'$ is the best approximation of $-F_0 + \sum_{i=1}^{m} y_i F_i \mid (y_1, \ldots, y_m)$.

3.5 Extensions and Alternative Implementation

As seen in §4, our algorithm tends to produce solutions with large numerators and denominators in the sum-of-square decomposition. We experimented with methods to get $F(\mathbf{y}') \approx F(\mathbf{y})$ such that $F(\mathbf{y}')$ has a smaller common denominator. This reduces to the following problem: given $\mathbf{v} \in \mathbf{f}_0 + \mathrm{vect}(\mathbf{f}_1, \ldots, \mathbf{f}_n)$ a real (floating-point) vector and $\mathbf{f}_0, \ldots, \mathbf{f}_n$ rational vectors, find $\mathbf{y}'_1, \ldots, \mathbf{y}_n$ such that $\mathbf{v}' = \mathbf{f}_0 + \sum_i y'_i \mathbf{f}_i \approx \mathbf{v}$ and the numerators of $\mathbf{v}'$ have a tunable magnitude (parameter μ). One can obtain such a result by LLL reduction of the rows of:

$$M = \begin{pmatrix} \mathbb{Z}(\beta\mu(\mathbf{f}_0 - \mathbf{v})) & \mathbb{Z}(\beta\mathbf{f}_0) & 1 & 0 & \ldots & 0 \\ \mathbb{Z}(\beta\mu\mathbf{f}_1) & \mathbb{Z}(\beta\mathbf{f}_1) & 0 & 1 & \ldots & \\ \vdots & \vdots & 0 & & \ddots & \\ \mathbb{Z}(\beta\mu\mathbf{f}_n) & \mathbb{Z}(\beta\mathbf{f}_n) & 0 & & & 1 \end{pmatrix} \tag{6}$$

where β is a large parameter (say, 10^{19}) and $\mathbb{Z}(v)$ stands for the integer rounding of v. After LLL reduction, one of the short vectors in the basis will be a combination $\sum_i^n y_i l_i$ where $l_0, \ldots, l_n$ are the rows of M, such that $y_0 \neq 0$. Because of the large $\beta\mu$ coefficient, $y_0(\mathbf{f}_0 - \mathbf{v}) + \sum_{i=1}^n y_i \mathbf{f}_i$ should be very small, thus $f_0 + \sum_{i=1}^n y_i \mathbf{f}_i \approx v$. But among those vectors, the algorithm chooses one such that $\sum_{i=0}^n y_i \mathbf{f}_i$ is not large — and among the suitable v', the vector of numerators is proportional to $\sum_{i=0}^n y_i \mathbf{f}_i$.

After computing such a y', we check whether $F(\mathbf{y}') \succeq 0$; we try this for a geometrically increasing sequence of μ and stop as soon as we find a solution. The matrices $\hat{Q}_j$ then have simpler coefficients than the original ones. Unfortunately, it does not ensue that the sums of square decompositions of these matrices have small coefficients.

An alternative to finding some kernel vectors of a single matrix would be to compute several floating-point matrices, for instance obtained by SDP solving with optimization in multiple directions, and find common kernel vectors using LLL.

3.6 Sub-algorithms and Implementation

The reduction from the problem expressed in Eq. 3 to SDP with rational solutions was implemented in Sage.[7]

Solving the systems of linear equations (S) (Sec. 3.1, over the coefficients of the matrices) and 5, in order to obtain a system $-F_0 + \mathrm{vect}(F_1, \ldots, F_m)$ of generators of the solution space, is done by echelonizing the equation system (in homogeneous form) in exact arithmetic, then reading the solution off the echelon form. The dimension of the system is quadratic in the number of monomials (on the problems we experimented with, dimensions up to 7900 were found); thus efficient algorithms should be used. In particular, sparse Gaussian elimination

[7] Sage is a computer algebra system implemented using the Python programming language, available under the GNU GPL from `http://www.sagemath.org`

in rational arithmetic, which we initially experimented, is not efficient enough; we thus instead use a sparse multi-modular algorithm [28, ch. 7] from LinBox[8]. Multi-modular methods compute the desired result modulo some prime numbers, and then reconstruct the exact rational values.

One can test whether a symmetric rational matrix Q is positive semidefinite by attempting to convert it into its Gaussian decomposition, and fail once one detects a negative diagonal element, or a nonzero row with a zero diagonal element (Appendix. A). We however experimented with three other methods that perform better:

- Compute the minimal polynomial of Q using a multi-modular algorithm [1]. The eigenvalues of Q are its roots; one can test for the presence of negative roots using Descartes' rule of signs. Our experiments seem to show this is the fastest exact method.
- Compute the characteristic polynomial of Q using a multimodular algorithm [1] and do as above. Somewhat slower but more efficient than Gaussian decomposition.
- Given a basis B of the span of Q, compute the Cholesky decomposition of B^TQB by a numerical method. This decomposition fails if and only if B^TQB is not positive definite (up to numerical errors), thus succeeds if and only if Q is positive semidefinite (up to numerical errors).

  For efficiency, instead of computing the exact basis B of the span of Q, we use B from §3.3, whose span includes the span of Q. The only risk is that $\ker B \subsetneq \ker Q$ while Q is positive semidefinite, in which case B^TQB will have nontrivial nullspace and thus will be rejected by the Cholesky decomposition. This is not a problem in our algorithm: it just means that the procedure for finding kernel vectors by LLL will find vectors in $\ker Q \setminus \ker B$.

  One problem could be that the Cholesky decomposition will incorrectly conclude that B^TQB is not positive definite, while it is but has very small positive eigenvalues. In this case, our algorithm may then find kernel vectors that are not really kernel vectors, leading to an overconstrained system and possibly loss of completeness. We have not encountered such cases.

  Another problem could be that a Cholesky decomposition is obtained from a matrix not positive semidefinite, due to extremely bad numerical behavior. At worst, this will lead to rejection of the witness when the allegedly semidefinite positive matrices get converted to sums of squares, at the end of the algorithm.

Numerical SDP solving is performed using DSDP[9] [3, 4], communicating using text files. LLL reduction is performed by fpLLL.[10] Least square projection is performed using Lapack's DGELS.

[8] LinBox is a library for exact linear arithmetic, used by Sage for certain operations. `http://www.linalg.org/`

[9] DSDP is a sdp tool available from `http://www.mcs.anl.gov/DSDP/`

[10] fpLLL is a LLL library from Damien Stehlé et al., available from `http://perso.ens-lyon.fr/damien.stehle/`

The implementation is available from the first author's Web page (`http://bit.ly/fBNLhR` and `http://bit.ly/gPXNF8`).

3.7 Preliminary Reductions

The more coefficients to find there are, the higher the dimension is, the longer computation times grow and the more likely numerical problems become. Thus, any cheap technique that reduces the search space is welcome.

If one looks for witnesses for problems involving only homogeneous polynomials, then one can look for witnesses built out of a homogeneous basis of monomials (this technique is implemented in our positivity checker).

One could also make use of symmetries inside the problem. For instance, if one looks for a nonnegativity witness $P = N/D$ of a polynomial P, and P is symmetric (that is, there exists a substitution group Σ for the variables of P such that $P.\sigma = P$ for $\sigma \in \Sigma$), then one may reduce the search to symmetric N and D. If $P = N/D$ is a witness, then $DP = N$ thus for any σ, $(D.\sigma)P = (N.\sigma)$ and thus $(\sum_\sigma D.\sigma)P = (\sum_\sigma N.\sigma)$, thus $D' = \sum_\sigma D.\sigma$ and $N' = \sum_\sigma N.\sigma$ constitute a symmetric nonnegativity witness.

4 Examples

The following system of inequalities has no solution (neither Redlog nor QepCad nor Mathematica 5 can prove it; Mathematica 7 can):

$$\begin{cases} P_1 = x^3 + xy + 3y^2 + z + 1 \geq 0 \\ P_2 = 5z^3 - 2y^2 + x + 2 \geq 0 \quad P_3 = x^2 + y - z \geq 0 \\ P_4 = -5x^2z^3 - 50xyz^3 - 125y^2z^3 + 2x^2y^2 + 20xy^3 + 50y^4 - 2x^3 \\ \quad -10x^2y - 25xy^2 - 15z^3 - 4x^2 - 21xy - 47y^2 - 3x - y - 8 \geq 0 \end{cases} \tag{7}$$

This system was concocted by choosing P_1, P_2, P_3 somewhat haphazardly and then $P_4 = -(P_1 + (3 + (x + 5y)^2)P_2 + P_3 + 1 + x^2)$, which guaranteed the system had no solution. The initial 130 constraints yield a search space of dimension 145, and after four round of numeric solving one gets an unsatisfiability witness (sums of squares Q_j such that $\sum_{j=1}^{4} P_j Q_j + Q_5 = 0$). Total computation time was 4.4 s. Even though there existed a simple solution (note the above formula for P_4), our algorithm provided a lengthy one, with large coefficients (and thus unfit for inclusion here).

Motzkin's polynomial M (Eq. 1) cannot be expressed as a sum of squares, but it can be expressed as a quotient of two sums of squares. We solved $M.Q_1 - Q_2 = 0$ for sums of squares Q_1 and Q_2 built from homogeneous monomials of respective total degrees 3 and 6 — lesser degrees yield no solutions (Fig. 1). The equality relation over the polynomials yields 66 constraints over the matrix coefficients and a search space of dimension 186. Four cycles of SDP programming and LLL are then needed, total computation time was 4.1 s.

We exhibited witnesses that each of the 8 semidefinite positive forms listed by [24], which are not sums of squares of polynomials, are quotients of sums of

$Q_1 = 8006878A_1^2 + 29138091A_2^2 + 25619868453870/4003439A_3^2 + 14025608A_4^2 + 14385502A_5^2$
$+ 85108577038951965167/12809934226935A_6^2$
$Q_2 = 8006878B_1^2 + 25616453B_2^2 + 108749058736871/4003439B_3^2 + 161490847987681$
$/25616453B_4^2 + 7272614B_5^2 + 37419351B_6^2 + 13078817768190/3636307B_7^2 + 71344030945385471151$
$/15535579819553B_8^2 + 539969700325922707586/161490847987681B_9^2 + 41728880843834$
$/12473117B_{10}^2 + 131008857208463018914/62593321265751B_1 1^2$, where
$A_1 = -1147341/4003439x_1^2x_3 - 318460/4003439x_2^2x_3 + x_3^3$ $A_2 = x_2x_3^2$ $A_3 = -4216114037644$
$/12809934226935x_1^2x_3 + x_2^2x_3$ $A_4 = x_1x_3^2$, $A_5 = x_1x_2x_3$, $A_6 = x_1^2x_3$ and $B_1 = -1102857$
$/4003439x_1^4x_2x_3 - 5464251/4003439x_1^2x_2x_3^3 + 2563669/4003439x_2^3x_3^3 + x_2x_3^5$, $B_2 = -9223081$
$/25616453x_1^4x_3^2 - 18326919/25616453x_1^2x_2^2x_3^2 + 1933547/25616453x_2^4x_3^2 + x_2^2x_3^4$,
$B_3 = -2617184886847/15535579819553x_1^4x_2x_3 - 12918394932706/15535579819553x_1^2x_2x_3^3 + x_2^3x_3^3$,
$B_4 = -26028972147097/161490847987681x_1^4x_3^2 - 135461875840584$
$/161490847987681x_1^2x_2^2x_3^2 + x_2^4x_3^2$, $B_5 = -2333331/3636307x_1^3x_2x_3^2 - 1302976$
$/3636307x_1x_2^3x_3^2 + x_1x_2x_3^4$, $B_6 = -11582471/37419351x_1^5x_3 - 12629854$
$/37419351x_1^3x_2^2x_3 - 4402342/12473117x_1^3x_3^3 + x_1x_2^2x_3^3$, $B_7 = -x_1^3x_2x_3^2 + x_1x_2^3x_3^2$,
$B_8 = -x_1^4x_2x_3 + x_1^2x_2x_3^3$, $B_9 = -x_1^4x_3^2 + x_1^2x_2^2x_3^2$, $B_{10} = -17362252580967/20864440421917x_1^5x_3$
$- 3502187840950/20864440421917x_1^3x_2^2x_3 + x_1^3x_3^3$, $B_{11} = -x_1^5x_3 + x_1^3x_2^2x_3$.

Fig. 1. Motzkin's polynomial M (Eq. 1) as Q_2/Q_1

squares (Motzkin's M, Robinson's R and f, Choi and Lam's F, Q, S, H and Schmüdgen's q). These examples include polynomials with up to 6 variables and search spaces up to dimension 1155. We did likewise with *delzell*, *laxlax* and *leepstarr2* from [14]. The maximal computation time was 7'.

We then converted these witnesses into Coq proofs of nonnegativity using a simple Sage script. These proofs use the `Ring` tactic, which checks for polynomial identity. Most proofs run within a few seconds, though *laxlax* takes 7'39" and Robinson's f 5'07"; the witness for *leepstarr2* is too large for the parser. We also exhibited a witness that the `Vor1` polynomial cited by [25] is a sum of squares.

John Harrison kindly provided us with a collection of 14 problems that his system [11] could not find witnesses for. These problems generally have the form $P_1 \geq 0 \wedge \cdots \wedge P_n \geq 0 \Rightarrow R \geq 0$. In order to prove such implication, we looked for witnesses consisting of sums of squares $(Q_1, \ldots, Q_n, Q_R)$,such that $\sum_j Q_j P_j + Q_R R = 0$ with $Q_R \neq 0$, and thus $R = \frac{\sum_j Q_j P_j}{Q_R}$. In some cases, it was necessary to use the products $\prod_i P_i^{w_i}$ for $\mathbf{w} \in \{0,1\}^n$ instead of the P_i. We could find witnesses for all those problems,[11]. though for some of them, the witnesses are very large, taking up megabytes. Since these searches were done without making use of symmetries in the problem, it is possible that more clever techniques could find smaller witnesses.

5 Conclusion and Further Works

We have described a method for solving SDP problems in rational arithmetic. This method can be used to solve sums-of-squares problems even in geometrically degenerate cases. We illustrated this method with applications to proving the nonnegativity of polynomials, or the unsatisfiability of systems of polynomial (in) equalities. The method then provides easily checkable proof witnesses,

[11] A 7z archive is given at `http://bit.ly/hM7HW3`

in the sense that checking the witness only entails performing polynomial arithmetic and applying a few simple mathematical lemmas. We have implemented the conversion of nonnegativeness witnesses to Coq proofs. A more ambitious implementation, mapping Coq real arithmetic proofs goals to Positivstellensatz problems through the `Psatz` tactic from the MicroMega package [5], then mapping Positivstellensatz witnesses back to proofs, is underway.

One weakness of the method is that it tends to provide "unnatural" witnesses — they tend to have very large coefficients. These are machine-checkable but provide little insights to the reader. An alternative would be to provide the matrices and some additional data (such as their minimal polynomial) and have the checker verify that they are semidefinite positive; but this requires formally proving, once and for all, some non-trivial results on polynomials, symmetric matrices and eigenvalues (e.g. the Cayley-Hamilton theorem), as well as possibly performing costly computations, e.g. evaluating a matrix polynomial.

A more serious limitation for proofs of unsatisfiability is the very high cost of application of the Positivstellensatz. There is the exponential number of polynomials to consider, and the unknown number of monomials. It would be very interesting if there could be some simple results, similar to the Newton polytope approach, for reducing the dimension of the search space or the number of polynomials to consider. Another question is whether it is possible to define SDP problems from Positivstellensatz equations for which the spectrahedron has rational points only at its relative boundary.

While our method performed well on examples, and is guaranteed to provide a correct answer if it provides one, we have supplied no completeness proof — that is, we have not proved that it necessarily provides a solution if there is one. This is due to the use of floating-point computations. One appreciable result would be that a solution should be found under the assumption that floating-point computations are precise up to ϵ, for a value of ϵ and the various scaling factors in the algorithm depending on the values in the problem or the solution.

It seems possible to combine our reduction method based on LLL with the Newton iterations suggested by [13, 14], as an improvement over their strategy for detection of useless monomials and reduction of the search space. Again, further experiment is needed.

References

1. Adams, J., Saunders, B.D., Wan, Z.: Signature of symmetric rational matrices and the unitary dual of Lie groups. In: ISSAC 2005, pp. 13–20. ACM, New York (2005)
2. Basu, S., Pollack, R., Roy, M.-F.: On the combinatorial and algebraic complexity of quantifier elimination. Journal of the ACM (JACM) 43(6), 1002–1045 (1996)
3. Benson, S.J., Ye, Y.: DSDP5 user guide — software for semidefinite programming. technical memorandum 277. Argonne National Laboratory (2005)
4. Benson, S.J., Ye, Y.: Algorithm 875: DSDP5—software for semidefinite programming. ACM Transactions on Mathematical Software 34(3), 16:1–16:20 (2008)
5. Besson, F.: Fast reflexive arithmetic tactics the linear case and beyond. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 48–62. Springer, Heidelberg (2007)

6. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2004), `http://www.stanford.edu/~boyd/cvxbook/`
7. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Brakhage, H. (ed.) Automata Theory and Formal Languages. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
8. Collins, G.E.: Quantifier elimination by cylindrical algebraic decomposition — twenty years of progress. In: Caviness, B.F., Johnson, J.R. (eds.) Quantifier Elimination and Cylindrical Algebraic Decomposition, pp. 8–23 (1998)
9. Coste, M., Lombardi, H., Roy, M.F.: Dynamical method in algebra: effective Nullstellensätze. Annals of Pure and Applied Logic 111(3), 203–256 (2001); Proceedings of the International Conference "Analyse & Logique" Mons, Belgium, August 25-29 (1997)
10. Dutertre, B., de Moura, L.: Integrating simplex with DPLL(T). Tech. Rep. SRI-CSL-06-01, SRI International (May 2006)
11. 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)
12. INRIA: The Coq Proof Assistant Reference Manual, 8.1 edn. (2006)
13. Kaltofen, E., Li, B., Yang, Z., Zhi, L.: Exact certification of global optimality of approximate factorizations via rationalizing sums-of-squares with floating point scalars. In: ISSAC (International Symposium on Symbolic and Algebraic Computation), pp. 155–163. ACM, New York (2008)
14. Kaltofen, E., Li, B., Yang, Z., Zhi, L.: Exact certification in global polynomial optimization via sums-of-squares of rational functions with rational coefficients (2009), accepted at J. Symbolic Computation
15. de Klerk, E., Pasechnik, D.V.: Products of positive forms, linear matrix inequalities, and Hilbert 17th problem for ternary forms. European Journal of Operational Research, 39–45 (2004)
16. Krivine, J.L.: Anneaux préordonnés. J. Analyse Math. 12, 307–326 (1964), `http://hal.archives-ouvertes.fr/hal-00165658/en/`
17. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
18. Lombardi, H.: Théorème des zéros réels effectif et variantes. Tech. rep., Université de Franche-Comté, Besan con, France (1990), `http://hlombardi.free.fr/publis/ThZerMaj.pdf`
19. Lombardi, H.: Une borne sur les degrés pour le théorème des zéros réel effectif. In: Coste, M., Mahé, L., Roy, M.F. (eds.) Real Algebraic Geometry: Proceedings of the Conference held in Rennes, France. Lecture Notes in Mathematics, vol. 1524, Springer, Heidelberg (1992)
20. Mahboubi, A.: Implementing the CAD algorithm inside the Coq system. Mathematical Structures in Computer Sciences 17(1) (2007)
21. Parrilo, P.: Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization. Ph.D. thesis, California Institute of Technology (2000)
22. Peyrl, H., Parrilo, P.A.: Computing sum of squares decompositions with rational coefficients. Theoretical Computer Science 409(2), 269–281 (2008)
23. Reznick, B.: Extremal PSD forms with few terms. Duke Mathematical Journal 45(2), 363–374 (1978)
24. Reznick, B.: Some concrete aspects of Hilbert's 17th problem. In: Delzell, C.N., Madden, J.J. (eds.) Real Algebraic Geometry and Ordered Structures, Contemporary Mathematics, vol. 253. American Mathematical Society, Providence (2000)

25. Safey El Din, M.: Computing the global optimum of a multivariate polynomial over the reals. In: ISSAC (International Symposium on Symbolic and Algebraic Computation), pp. 71–78. ACM, New York (2008)
26. Schweighofer, M.: On the complexity of Schmüdgen's Positivstellensatz. Journal of Complexity 20(4), 529–543 (2004)
27. Seidenberg, A.: A new decision method for elementary algebra. Annals of Mathematics 60(2), 365–374 (1954), http://www.jstor.org/stable/1969640
28. Stein, W.A.: Modular forms, a computational approach. Graduate studies in mathematics. American Mathematical Society, Providence (2007)
29. Stengle, G.: A Nullstellensatz and a Positivstellensatz in semialgebraic geometry. Mathematische Annalen 2(207), 87–87 (1973)
30. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press, Berkeley (1951)
31. Tiwari, A.: An algebraic approach for the unsatisfiability of nonlinear constraints. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 248–262. Springer, Heidelberg (2005)
32. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Review 38(1), 49–95 (1996)

A Gaussian Reduction and Positive Semidefiniteness

An algorithm for transforming a semidefinite positive matrix into a "sum of squares" form, also known as Gaussian reduction:

```
for i:=1 to n do
begin
  if m[i, i] < 0 then
    throw non_positive_semidefinite
  if m[i, i] = 0 then
    if m.row(i) <> 0
      throw non_positive_semidefinite
  else
    begin
      v := m.row(i) / m[i, i]
      output.append(m[i, i], v)
     m := m − m[i, i] ∗ v.transpose() ∗ v
    end
end
```

Suppose that the entrance of iteration i, $m[i \dots n, i \dots n]$ is positive semidefinite. If $m_{i,i} = 0$, then the ith base vector is in the isotropic cone of the matrix, thus of its kernel, and the row i must be zero. Otherwise, $m_{i,i} > 0$. By adding ϵ to the diagonal of the matrix, we would have a positive definite matrix and thus the output of the loop iteration would also be positive definite, as above. By $\epsilon \to 0$ and the fact that the set of positive semidefinite matrices is topologically closed, then the output of the loop iteration is also positive semidefinite.

The output variable is then a list of couples (c_i, v_i) such that $c_i > 0$ and the original matrix m is equal to $\sum_i c_i v_i^T v_i$ (with v_i row vectors). Otherwise said, for any row vector u, $umu^T = \sum_i c_i \langle u, v_i \rangle^2$.

A Verified Runtime for a Verified Theorem Prover

Magnus O. Myreen[1] and Jared Davis[2]

[1] Computer Laboratory, University of Cambridge, UK
[2] Centaur Technology, Inc., Austin TX, USA

Abstract. Theorem provers, such as ACL2, HOL, Isabelle and Coq, rely on the correctness of runtime systems for programming languages like ML, OCaml or Common Lisp. These runtime systems are complex and critical to the integrity of the theorem provers.

In this paper, we present a new Lisp runtime which has been formally verified and can run the Milawa theorem prover. Our runtime consists of 7,500 lines of machine code and is able to complete a 4 gigabyte Milawa proof effort. When our runtime is used to carry out Milawa proofs, less unverified code must be trusted than with any other theorem prover.

Our runtime includes a just-in-time compiler, a copying garbage collector, a parser and a printer, all of which are HOL4-verified down to the concrete x86 code. We make heavy use of our previously developed tools for machine-code verification. This work demonstrates that our approach to machine-code verification scales to non-trivial applications.

1 Introduction

We can never be sure [6] a computer has executed a theorem prover (or any other program) correctly. Even if we could prove a processor design implements its instruction set, we have no way to ensure it will be manufactured correctly and will not be interfered with as it runs. But can we develop a theorem prover for which there are no other reasonable doubts?

Any theorem prover is based on a formal mathematical logic. Logical soundness is well-studied. It is usually established with social proofs, but some soundness proofs [20,10] have even been checked by computers. If we accept the logic is sound, the question boils down to whether the theorem prover is *faithful* to its logic: does it only claim to prove formulas that are indeed theorems?

In many theorem provers, the trusted core—the code that must be right to ensure faithfulness—is quite small. As examples, HOL Light [12] is an LCF-style system whose trusted core is 400 lines of Objective Caml, and Milawa [5] is a Boyer-Moore style prover whose trusted core is 2,000 lines of Common Lisp. These cores are so simple we may be able to prove their faithfulness socially, or perhaps even mechanically as Harrison [11] did for HOL Light.

On the other hand, to actually use these theorem provers we need a runtime environment that can parse source code, infer types, compile functions, collect garbage, and so forth. These runtimes are far more complicated than simple

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 265–280, 2011.

theorem-prover cores. For a rough perspective, source-code distributions of Objective Caml and Common Lisp systems seem to range from 15 MB to 50 MB on disk, and also require C compilers and various libraries.

In this paper, we present *Jitawa*, the first mechanically verified runtime designed to run a general-purpose theorem prover.

– We target the Milawa theorem prover, so we begin with a brief description of this system and explain how using a verified runtime increases our level of trust in Milawa proofs. (Section 2)
– To motivate the design of our runtime, we examine Milawa's computational and I/O needs. To meet these needs, Jitawa features efficient parsing, just-in-time compilation to 64-bit x86 machine code, garbage collection, expression printing, and an "abort with error message" capability. (Section 3)
– We consider what it means for Jitawa to be correct. We develop a formal HOL4 [21] specification (400 lines) of how the runtime should operate. This covers expression evaluation, parsing, and printing. (Section 4)
– We explain how Jitawa is implemented and verified. We build heavily on our previous tools for machine-code synthesis and verification, so in this paper we focus on how our compiler is designed and specified and also on how I/O is handled. We present the top-level correctness theorem that shows Jitawa's machine code implements its specification. (Section 5)
– We describe the relationship between Milawa and Jitawa. We have used Jitawa to carry out a 4 GB proof effort in Milawa, demonstrating the good capacity of the runtime. We explain the informal nature of this connection, and how we hope it may be formalized in future work. (Section 6)

We still need *some* unverified code. We have not tried to avoid using an operating system, and we use a C wrapper-program to interact with it. This C program is quite modest: it uses `malloc` to allocate memory and invokes our runtime. Jitawa also performs I/O by making calls to C functions for reading and writing standard input and output (Section 5.3).

2 The Milawa System

Milawa [5] is a theorem prover styled after systems like NQTHM [1] and ACL2 [13]. The Milawa logic has three kinds of objects: natural numbers, symbols, and conses. It also has twelve primitive functions like if, equal, cons, and $+$, and eleven macros like list, and, let*, and cond. Starting from these primitives, one may introduce the definitions of first-order, total, untyped, recursive functions as axioms. For instance, a list-length function might be introduced as

$$\forall \texttt{x.}\ \texttt{(len x)} = \texttt{(if (consp x) (+ '1 (len (cdr x))) '0)}.$$

Almost all of Milawa's source code is written as functions in its logic. We can easily run these functions on a Common Lisp system.

2.1 The Trusted Core

Milawa's original trusted core is a 2,000 line Common Lisp program that checks a file of *events*. Most commonly,

- `Define` events are used to introduce recursive functions, and include the name of a file that should contain a proof of termination, and
- `Verify` events are used to admit formulas as theorems, and include the name of a file that should contain a proof of the formula.

The user generates these events and proof files ahead of time, with the help of an interactive interface that need not be trusted.

A large part of the trusted core is just a straightforward definition of formulas and proofs in the Milawa logic. A key function is proofp ("proof predicate"), which determines if its argument is a valid Milawa-logic proof; this function only accepts full proofs made up of primitive inferences, and it is defined in the Milawa logic so we can reason about provability.

When the trusted core is first started, proofp is used to check the proofs for each event. But, eventually, the core can be reflectively extended. The steps are:

1. `Define` a new proof-checking function. This function is typically a proper extension of proofp: it still accepts all the primitive proof steps, but it also permits new, non-primitive proof steps.
2. `Verify` that the new function only accepts theorems. That is, whenever the new proof checker accepts a proof of some formula ϕ, there must exist a proof of ϕ that is accepted by proofp.
3. Use the special `Switch` event to instruct the trusted core to begin using the new, now-verified function, instead of proofp, to check proofs.

After such an extension, the proofs for `Define` and `Verify` events may make use of the new kinds of proof steps. These higher-level proofs are usually much shorter than full proofp-style proofs, and can be checked more quickly.

2.2 The Verified Theorem Prover

Milawa's trusted core has no automation for finding proofs. But separately from its core, Milawa includes a Boyer-Moore style theorem prover that can carry out a goal-directed proof search using algorithms like lemma-driven conditional rewriting, calculation, case-splitting into subgoals, and so on.

All of these algorithms are implemented as functions in the Milawa logic. Because of this, we can reason about their behavior using the trusted core. In Milawa's "self-verification" process, the trusted core is used to `Define` each of the functions making up the theorem prover and `Verify` lemmas about their behavior. This process culminates in the definition of a verified proof checker that can apply any sequence of Milawa's tactics as a single proof step. Once we `Switch` to this new proof checker, the trusted core can essentially check proofs by directly running the theorem prover.

2.3 The Role of a Verified Runtime

Through its self-verification process, the Milawa theorem prover is mechanically verified by its trusted core. For this verification to be believed—indeed, for any theorems proven by Milawa to be believed—one must trust that

1. the Milawa logic is sound,
2. the trusted core of Milawa is faithful to its logic, and
3. the computing platform used to carry out the self-verification process has correctly executed Milawa's trusted core.

The first two points are addressed in previous work [5] in a social way. Our verified runtime does not directly bolster these arguments, but may eventually serve as groundwork for a mechanical proof of these claims (Section 6).

The third point requires trusting some computer hardware and a Common Lisp implementation. Unfortunately, these runtimes are always elaborate and are never formally verified. Using Jitawa as our runtime greatly reduces the amount of unverified code that must be trusted.

3 Requirements and Design Decisions

On the face of it, Milawa is quite modest in what it requires of the underlying Lisp runtime. Most of the code for its trusted core and all of the code for its theorem prover are written as functions in the Milawa logic. These functions operate on just a few predefined data types (natural numbers, symbols, and conses), and involve a handful of primitive functions and macros like car, +, list, and cond. To run these functions we just need a basic functional programming language that implements these primitives.

Beyond this, Milawa's original trusted core also includes some Common Lisp code that is outside of the logic. As some examples:

- It destructively updates global variables that store its arity table, list of axioms, list of definitions, and so forth.
- It prints some status messages and timing information so the user can evaluate its progress and performance.
- It can use the underlying Lisp system's checkpointing system to save the program's current state as a new executable.

It was straightforward to develop a new version of the Milawa core that does away with the features mentioned above: we avoid destructive updates by adopting a more functional "state-tuple" style, and simply abandon checkpointing and timing reports since, while convenient, they are not essential.

On the other hand, some other Common Lisp code is not so easy to deal with. In particular:

- It instructs the Common Lisp system to compile user-supplied functions as they are `Define`d, which is important for running new proof checkers.

- It dynamically calls either proofp or whichever proof checker has been most recently installed via Switch to check proofs.
- It aborts with a runtime error when invalid events or proofs are encountered, or if an attempt is made to run a Skolem function.

We did not see a good way to avoid any of this. Accordingly, Jitawa must also provide on-the-fly compilation of user-defined functions, dynamic function invocation, and some way to cause runtime errors.

3.1 I/O Requirements

In Milawa's original trusted core, each Define and Verify event includes the name of a file that should contain the necessary proof, and these files are read on demand as each event is processed. For a rough sense of scale, the proof of self-verification is a pretty demanding effort; it includes over 15,000 proof files with a total size of 8 GB.

The proofs in these files—especially the lowest-level proofs that proofp checks—can be very large and repetitive. As a simple but crucial optimization, an abbreviation mechanism [2] lets us reuse parts of formulas and proofs. For instance,

```
(append (cons (cons a b) c)
        (cons (cons a b) c))
```

could be more compactly written using an abbreviation as

```
(append #1=(cons (cons a b) c)
        #1#).
```

We cannot entirely avoid file input since, at some point, we must at least tell the program what we want it to verify. But we would prefer to minimize interaction with the operating system. Accordingly, in our new version of the Milawa core, we do not keep proofs in separate files. Instead, each event directly contains the necessary proof, so we only need to read a single file. This approach exposes additional opportunities for structure sharing. While the original, individual proof files for the bootstrapping process are 8 GB, the new events file is only 4 GB. It has 525 million abbreviations.

At any rate, Jitawa needs to be able to parse input files that are gigabytes in size and involve hundreds of millions of abbreviations.

3.2 Designing for Performance and Scalability

The real challenge in constructing a practical runtime for Milawa (or any other theorem prover) is that performance and scalability cannot be ignored. Our previously verified Lisp interpreter [18] is hopelessly inadequate: its direct interpreter approach is too slow, and it also has inherent memory limitations that prevent it from handling the large objects the theorem prover must process.

For Jitawa, we started from scratch and made sure the central design decisions allowed our implementation to scale. For instance:

- To improve execution times, functions are just-in-time compiled to native x86 machine code.
- To support large computations, we target 64-bit x86. Jitawa can handle up to 2^{31} live cons cells, i.e., up to 16 GB of conses at 8 bytes per cons.
- Parsing and printing are carefully coded not to use excessive amounts of memory. In particular, lexing is merged with parsing into what is called a scanner-less parser, and abbreviations are supported efficiently.
- Since running out of heap space or stack space is a real concern, we ensure graceful exits in all circumstances and provide helpful error messages when limits are reached.

4 The Jitawa Specification

Jitawa implements a read-eval-print loop. Here is an example run, where lines starting with > are user input and the others are the output.

```
> ’3
3
> (cons ’5 ’(6 7))
(5 6 7)
> (define ’increment ’(n) ’(+ n ’1))
NIL
> (increment ’5)
6
```

What does it mean for Jitawa to be correct? Intuitively, we need to show the input characters are parsed as expected, the parsed terms are evaluated according to our intended semantics, and the results of evaluation are printed as the correct character sequences.

To carry out a proof of correctness, we first need to formalize how parsing, evaluation, and printing are supposed to occur. In this section, we describe our formal, HOL specification of how Jitawa is to operate. This involves defining a term representation and evaluation semantics (Sections 4.1 and 4.2), and specifying how parsing and printing (Section 4.3) are to be done. We combine these pieces into a top-level specification (Section 4.4) for Jitawa.

Altogether, our specification takes 400 lines of HOL code. It is quite abstract: it has nothing to do with the x86 model, compilation, garbage collection, and so on. We eventually (Section 5.4) prove Jitawa's machine code implements this specification, and we regard this as a proof of "Jitawa is correct."

4.1 Syntax

Milawa uses a typical s-expression [15] syntax. While Jitawa's parser has to deal with these expressions at the level of individual characters, it is easier to model these expressions as a HOL datatype,

$$\begin{array}{rcll} \mathit{sexp} & ::= & \mathsf{Val}\ \mathit{num} & \text{(natural numbers)} \\ & | & \mathsf{Sym}\ \mathit{string} & \text{(symbols)} \\ & | & \mathsf{Dot}\ \mathit{sexp}\ \mathit{sexp} & \text{(cons pairs).} \end{array}$$

We use the name Dot instead of Cons to distinguish it from the name of the function called Cons which produces this structure; the name Dot is from the syntax `(1 . 2)`. As an example, the *sexp* representation of `(+ n '1)` is

```
Dot (Sym "+")
    (Dot (Sym "N")
         (Dot (Dot (Sym "QUOTE") (Dot (Val 1) (Sym "NIL")))
              (Sym "NIL"))).
```

Our specification also deals with well-formed s-expression, i.e. s-expressions that can be evaluated. We represent these expressions with a separate datatype, called *term*. The *term* representation of `(+ n '1)` is

$$\mathsf{App}\ (\mathsf{PrimitiveFun\ Add})\ [\mathsf{Var}\ \texttt{"N"}, \mathsf{Const}\ (\mathsf{Val}\ 1)].$$

The definition of *term* is shown below. Some constructors are marked as macros, meaning they expand into other terms in our semantics and in the compiler, e.g., Cond expands into If (if-then-else) statements. These are the same primitives and macros as in the Milawa theorem prover.

$$\begin{array}{lcll}
\textit{term} & ::= & \mathsf{Const}\ \textit{sexp} & \\
 & \mid & \mathsf{Var}\ \textit{string} & \\
 & \mid & \mathsf{App}\ \textit{func}\ (\textit{term}\ \mathsf{list}) & \\
 & \mid & \mathsf{If}\ \textit{term}\ \textit{term}\ \textit{term} & \\
 & \mid & \mathsf{LambdaApp}\ (\textit{string}\ \mathsf{list})\ \textit{term}\ (\textit{term}\ \mathsf{list}) & \\
 & \mid & \mathsf{Or}\ (\textit{term}\ \mathsf{list}) & \\
 & \mid & \mathsf{And}\ (\textit{term}\ \mathsf{list}) & \text{(macro)} \\
 & \mid & \mathsf{List}\ (\textit{term}\ \mathsf{list}) & \text{(macro)} \\
 & \mid & \mathsf{Let}\ ((\textit{string} \times \textit{term})\ \mathsf{list})\ \textit{term} & \text{(macro)} \\
 & \mid & \mathsf{LetStar}\ ((\textit{string} \times \textit{term})\ \mathsf{list})\ \textit{term} & \text{(macro)} \\
 & \mid & \mathsf{Cond}\ ((\textit{term} \times \textit{term})\ \mathsf{list}) & \text{(macro)} \\
 & \mid & \mathsf{First}\ \textit{term} \mid \mathsf{Second}\ \textit{term} \mid \mathsf{Third}\ \textit{term} & \text{(macro)} \\
 & \mid & \mathsf{Fourth}\ \textit{term} \mid \mathsf{Fifth}\ \textit{term} & \text{(macro)} \\
 & & & \\
\textit{func} & ::= & \mathsf{Define} \mid \mathsf{Print} \mid \mathsf{Error} \mid \mathsf{Funcall} & \\
 & \mid & \mathsf{PrimitiveFun}\ \textit{primitive} \mid \mathsf{Fun}\ \textit{string} & \\
 & & & \\
\textit{primitive} & ::= & \mathsf{Equal} \mid \mathsf{Symbolp} \mid \mathsf{SymbolLess} & \\
 & \mid & \mathsf{Consp} \mid \mathsf{Cons} \mid \mathsf{Car} \mid \mathsf{Cdr} \mid & \\
 & \mid & \mathsf{Natp} \mid \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Less} &
\end{array}$$

4.2 Evaluation Semantics

We define the semantics of expressions as a relation $\xrightarrow{\mathsf{ev}}$ that explains how objects of type *term* evaluate. Following Common Lisp, we separate the store k for functions from the environment *env* for local variables. We model the I/O streams *io* as a pair of strings, one for characters produced as output, and one for characters yet to be read as input. Our evaluation relation $\xrightarrow{\mathsf{ev}}$ explains how terms may be evaluated with respect to some particular k, *env*, and *io* to produce a resulting *sexp* and an updated k' and io'.

As an example, the following rule shows how Var terms are evaluated. We only permit the evaluation of bound variables, i.e. $x \in \mathsf{domain}\ env$.

$$\frac{x \in \mathsf{domain}\ env}{(\mathsf{Var}\ x, env, k, io) \xrightarrow{\mathsf{ev}} (env(x), k, io)}$$

Our evaluation relation is defined inductively with auxilliary relations $\xrightarrow{\mathsf{evl}}$ for evaluating a list of terms and $\xrightarrow{\mathsf{ap}}$ for applying functions. For instance, the following rule explains how a function (i.e., something of type *func*) is applied: first the arguments are evaluated using $\xrightarrow{\mathsf{evl}}$, then the apply relation $\xrightarrow{\mathsf{ap}}$ determines the result of the application.

$$\frac{(args, env, k, io) \xrightarrow{\mathsf{evl}} (vals, k', io') \ \wedge\ (f, vals, env, k', io') \xrightarrow{\mathsf{ap}} (ans, k'', io'')}{(\mathsf{App}\ f\ \ args, env, k, io) \xrightarrow{\mathsf{ev}} (ans, k'', io'')}$$

With regards to just-in-time compilation, an interesting case for the apply relation $\xrightarrow{\mathsf{ap}}$ is the application of user-defined functions. In our semantics, a user-defined function *name* can be applied when it is defined in store k with the right number of parameters.

$$\frac{\begin{array}{c}k(name) = (params, body)\ \wedge\ (\mathsf{length}\ vals = \mathsf{length}\ params)\ \wedge \\ (body, [params \leftarrow vals], k, io) \xrightarrow{\mathsf{ev}} (ans, k', io')\ \wedge\ name \notin \mathsf{reserved\_names}\end{array}}{(\mathsf{Fun}\ name, vals, env, k, io) \xrightarrow{\mathsf{ap}} (ans, k', io')}$$

Another interesting case is how user-defined functions are introduced. New definitions can be added to k by evaluation of the Define function. We disallow overwriting existing definitions, i.e. $name \notin \mathsf{domain}\ k$.

$$\frac{name \notin \mathsf{domain}\ k}{(\mathsf{Define}, [name, params, body], env, k, io) \xrightarrow{\mathsf{ap}} (\mathsf{nil}, k[name \mapsto (params, body)], io)}$$

In Jitawa's implementation, an application of Define compiles the expression *body* into machine code. Notice how nothing in the above rule requires that it should be possible to evaluate the expression *body* at this stage. In particular, the functions mentioned inside *body* might not even be defined yet. This means that how we compile function calls within *body* depends on the compile-time state: if the function to be called is already defined we can use a direct jump/call to its code, but otherwise we use a slower, dynamic jump/call.

Strictly speaking, Milawa does not require that Define is to be applicable to functions that cannot be evaluated. However, we decided to allow such definitions to keep the semantics clean and simple. Another advantage of allowing compilation of calls to not-yet-defined functions is that we can immediately support mutually recursive definitions, e.g.:

```
(define 'even '(n) (if (equal n '0) 't   (odd  (- n '1))))
(define 'odd  '(n) (if (equal n '0) 'nil (even (- n '1))))
```

When the expression for even is compiled, the compiler knows nothing about the function odd and must thus insert a dynamic jump to the code for odd. But when odd is compiled, even is already known and the compiler can insert a direct jump to the code for even.

4.3 Parsing and Printing

Besides evaluation, our runtime must provide parsing and printing. We begin by modeling our parsing and printing algorithms at an abstract level in HOL as two functions, sexp2string and string2sexp, which convert s-expressions into strings and vice versa. The printing function is trivial. Parsing is more complex, but we can gain some assurance our specification is correct by proving it is the inverse of the printing function, i.e.

$$\forall s.\ \ \mathsf{string2sexp}\ (\mathsf{sexp2string}\ s)\ =\ s.$$

Unfortunately, Jitawa's true parsing algorithm must be slightly more complicated. It must handle the #1=-style abbreviations described in Section 3.1. Also, the parser we verified in previous work [18] assumed the entire input string was present in memory, but since Jitawa's input may be gigabytes in size, we instead want to read the input stream incrementally. We define a function,

$$\mathsf{next\_sexp}\ :\ \mathit{string} \rightarrow \mathit{sexp} \times \mathit{string},$$

that only parses the first s-expression from an input string and returns the unread part of the string to be read later.

We can prove a similar "inverse" theorem for next_sexp via a printing function, abbrevs2string, that prints a list of s-expressions, each using some abbreviations a. That is, we show next_sexp correctly reads the first s-expression, and leaves the other expressions for later:

$$\forall s\ a\ \mathit{rest}.\ \ \mathsf{next\_sexp}\ (\mathsf{abbrevs2string}\ ((s,a) :: \mathit{rest}))\ =\ (s, \mathsf{abbrevs2string}\ \mathit{rest}).$$

4.4 Top-Level Specification

We give our top-level specification of what constitutes a valid Jitawa execution as an inductive relation, $\xrightarrow{\mathsf{exec}}$. Each execution terminates when the input stream ends or contains only whitespace characters.

$$\frac{\mathsf{is\_empty}\ (\mathsf{get\_input}\ \mathit{io})}{(k, \mathit{io}) \xrightarrow{\mathsf{exec}} \mathit{io}}$$

Otherwise, the next s-expression is read from the input stream using next_sexp, this s-expression s is then evaluated according to $\xrightarrow{\mathsf{ev}}$, and finally the result of evaluation, ans, is appended to the output stream before execution continues.

$$\frac{\begin{array}{l}\neg\mathsf{is\_empty}\ (\mathsf{get\_input}\ io)\ \wedge \\ \mathsf{next\_sexp}\ (\mathsf{get\_input}\ io)) = (s, \mathit{rest})\ \wedge \\ (\mathsf{sexp2term}\ s, [], k, \mathsf{set\_input}\ \mathit{rest}\ io) \xrightarrow{\mathsf{ev}} (\mathit{ans}, k', io')\ \wedge \\ (k', \mathsf{append\_to\_output}\ (\mathsf{sexp2string}\ \mathit{ans})\ io') \xrightarrow{\mathsf{exec}} io''\end{array}}{(k, io) \xrightarrow{\mathsf{exec}} io''}$$

5 The Jitawa Implementation

The verified implementation of Jitawa is 7,500 lines of x86 machine code. Most of this code was not written and verified by hand. Instead, we produced the implementation using a combination of manual verification, decompilation and proof-producing synthesis [19].

1. We started by defining a simple stack-based bytecode language into which we can easily compile Lisp programs using a simple compilation algorithm.
2. Next, we defined a heap invariant and proved that certain machine instruction "snippets" implement basic Lisp operations and maintain this invariant.
3. These snippets of verified machine code were then given to our extensible synthesis tool [19] which we used to synthesise verified x86 machine code for our compilation algorithm.
4. Next, we proved the concrete byte representation of the abstract bytecode instructions is *in itself* machine code which performs the bytecode instructions themselves. Thus jumping directly to the concrete representation of the bytecode program will correctly execute it on the x86 machine.
5. Finally, we verified code for parsing and printing of s-expressions from an input and output stream and connected these up with compilation to produce a "parse, compile, jump to compiled code, print" loop, which we have proved implements Jitawa's specification.

Steps 2 and 3 correspond very closely to how we synthesised, in previous work [18], verified machine-code for our Lisp evaluation function lisp_eval.

5.1 Compilation to Bytecode

Jitawa compiles all expressions before they are executed. Our compiler targets a simple stack-based bytecode shown in Figure 1. At present, no optimizations are performed except for tail-call elimination and a simple optimization that speeds up evaluation of LambdaApp, Let and LetStar.

We model our compilation algorithm as a HOL function that takes the name, parameters, and body of the new function, and also a system state s. It returns a new system state, s', where the compiled code for body has been installed and other minor updates have been made.

$$\mathsf{compile}\ (\mathit{name}, \mathit{params}, \mathit{body}, s)\ =\ s'$$

bytecode	::=	Pop	pop one stack element
	|	PopN *num*	pop n stack elements below top element
	|	PushVal *num*	push a constant number
	|	PushSym *string*	push a constant symbol
	|	LookupConst *num*	push the nth constant from system state
	|	Load *num*	push the nth stack element
	|	Store *num*	overwrite the nth stack element
	|	DataOp *primitive*	add, subtract, car, cons, ...
	|	Jump *num*	jump to program point n
	|	JumpIfNil *num*	conditionally jump to n
	|	DynamicJump	jump to location given by stack top
	|	Call *num*	static function call (faster)
	|	DynamicCall	dynamic function call (slower)
	|	Return	return to calling function
	|	Fail	signal a runtime error
	|	Print	print an object to stdout
	|	Compile	compile a function definition

Fig. 1. Abstract syntax of our bytecode

We model the execution of bytecode using an operational semantics based on a next-state relation $\xrightarrow{\mathsf{next}}$. For simplicity and efficiency, we separate the value stack xs from the return-address stack rs; the relation also updates a program counter p and the system state s. The simplest example of $\xrightarrow{\mathsf{next}}$ is the Pop instruction, which just pops an element off the expression stack and advances the program counter to the next instruction.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Pop}])}{(top :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (xs, rs, p + \mathsf{length}(\mathsf{Pop}), s)}$$

Call *pos* is not much more complicated: we change the program counter to *pos* and push a return address onto the return stack.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Call}\ pos])}{(xs, rs, p, s) \xrightarrow{\mathsf{next}} (xs, (p + \mathsf{length}(\mathsf{Call}\ pos)) :: rs, pos, s)}$$

A DynamicCall is similar, but reads the name and expected arity n of the function to call from the stack, then searches in the current state to locate the position *pos* where the compiled code for this function begins.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{DynamicCall}])\ \wedge\ \mathsf{find\_func}\ (fn, s) = \mathsf{some}\ (n, pos)}{(\mathsf{Sym}\ fn :: \mathsf{Val}\ n :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (xs, (p + \mathsf{length}(\mathsf{DynamicCall})) :: rs, pos, s)}$$

The Print instruction is slightly more exotic: it appends the string representation of the top stack element, given by sexp2string (Section 4.3), onto the output stream, which is part of the system state s. It leaves the stack unchanged.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Print}])\ \wedge\ \mathsf{append\_to\_output}\ (\mathsf{sexp2string}\ top, s) = s'}{(top :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (top :: xs, rs, p + \mathsf{length}(\mathsf{Print}), s')}$$

The most interesting bytecode instruction is, of course, Compile. This instruction reads the name, parameter list, and body of the new function from the stack and updates the system state using the compile function.

$$\frac{\mathsf{contains\_bytecode}\ (p, s, [\mathsf{Compile}])\ \wedge\ \mathsf{compile}\ (name, params, body, s) = s'}{(body :: params :: name :: xs, rs, p, s) \xrightarrow{\mathsf{next}} (\mathsf{nil} :: xs, rs, p + \mathsf{length}(\mathsf{Compile}), s')}$$

At first sight, this definition might seem circular since the compile function operates over bytecode instructions. It is not circular: we first define the syntax of bytecode instructions, then the compile function which generates bytecode, and only then define the semantics of evaluating bytecode instructions, $\xrightarrow{\mathsf{next}}$.

Compile instructions are generated when we encounter an application of Define. For instance, when the compiler sees an expression like

```
(define 'increment '(n) '(+ n '1)),
```

it generates the following bytecode instructions (for some specific k):

PushSym "INCREMENT"	pushes symbol `increment` onto the stack
LookupConst k	pushes expression `(n)` onto the stack
LookupConst (k+1)	pushes expression `(+ n '1)` onto the stack
Compile	compiles the above expression

5.2 From Bytecode to Machine Code

Most compilers use some intermediate language before producing concrete machine code. However, our compiler goes directly from source to concrete machine code by representing bytecode instructions as a string of bytes that *are* machine code. For example, the Compile instruction is represented by bytes

```
48 FF 52 88
```

which happens to be 64-bit x86 machine code for `call [rdx-120]`, i.e. an instruction which makes a procedure call to a code pointer stored at memory address `rdx-120`.

For each of these byte sequences, we prove a *machine-code Hoare triple* [17] which states that it correctly implements the intended behaviour of the bytecode instruction in question with respect to a heap invariant lisp_bytecode_inv.

$$\begin{array}{l}
\mathsf{compile}\ (name, params, body, s) = s' \implies \\
\{\, \mathsf{lisp\_bytecode\_inv}\ (body :: params :: name :: xs, rs, s) * \mathsf{pc}\ p \,\} \\
\ p : \texttt{48 FF 52 D8} \\
\{\, \mathsf{lisp\_bytecode\_inv}\ (\mathsf{nil} :: xs, rs, s') * \mathsf{pc}\ (p + 4)\ \vee\ \mathsf{error} \,\}
\end{array}$$

At this stage you might wonder: but doesn't that Hoare triple rely on more code than just those four bytes? The answer is yes: it requires machine code which implements the compile function from above. We have produced such machine code by a method described in previous work [19]; essentially, we teach the theorem prover about basic operations w.r.t. to a heap invariant and then have the theorem prover synthesize machine code that implements the high-level algorithm for compilation. The code we synthesized in this way is part of lisp invariant lisp_bytecode_inv shown above. Thus when the above x86 instruction (i.e. `call [rdx-120]`) is executed, control just jumps to the synthesised code which when complete executes a return instruction that brings control back to the end of those four bytes.

The Hoare triples used here [17] do not require code to be at the centre of the Hoare triple as the following "code is data" theorem shows:

$$\forall p\ c\ q.\quad \{p\}\ c\ \{q\}\ =\ \{p * \mathsf{code}\ c\}\ \emptyset\ \{q * \mathsf{code}\ c\}$$

A detailed explanation of this rule, and a few others that are handy when dealing with self-modifying code, can be found in our previous paper [17].

5.3 I/O

Jitawa calls upon the external C routines `fgets` and `fputs` to carry out I/O. These external calls require assumptions in our proof. For instance, we assume that calling the routine at a certain location x—which our unverified C program initializes to the location of `fgets` before invoking the runtime—will:

1. produce a pointer z to a null-terminated string that contains the first n characters of the input stream, for some n, and
2. remove these first n characters from the input stream.

We further assume that the returned string is only empty if the input stream was empty. The machine-code Hoare triple representing this assumption is:

$$\begin{array}{l}\{\ \mathsf{rax}\ x * \mathsf{rbx}\ y * \mathsf{memory}\ m * \mathsf{io}\ (x, in, out) * \mathsf{pc}\ p\ \}\\ \ p : \mathtt{call\ rax}\\ \{\ \exists z\ n.\ \mathsf{rax}\ x * \mathsf{rbx}\ z * \mathsf{memory}\ m' * \mathsf{io}\ (x, \mathsf{drop}\ n\ in, out) * \mathsf{pc}\ (p+3)\ *\\ \qquad\qquad \langle \mathsf{string\_in\_mem\_at}\ (z, m', \mathsf{take}\ n\ in) \wedge (n = 0 \implies in = \mathtt{""})\rangle\ \}\end{array}$$

The fact that Jitawa implements an interactive read-eval-print loop is not apparent from our top-level correctness statement: it is just a consequence of reading lazily—our next_sexp style parser reads only the first s-expression, and `fgets` reads through at most the first newline—and printing eagerly.

5.4 Top-Level Correctness Theorem

The top-level correctness theorem is stated as the following machine-code Hoare triple. If the Jitawa implementation is started in a state where enough memory is allocated (init_state) and the input stream holds s-expressions for which an execution of Jitawa terminates, then either a final state described by $\xrightarrow{\mathsf{exec}}$ is reached or an error message is produced.

$$\begin{array}{l}\{\,\mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io\rangle\,\}\\ \ p : \mathsf{code\_for\_entire\_jitawa\_implementation}\\ \{\,\mathsf{error\_message} \vee \exists io'.\ \langle ([], io) \xrightarrow{\mathsf{exec}} io'\rangle * \mathsf{final\_state}\ io'\,\}\end{array}$$

This specification allows us to resort to an error message even if the evaluated s-expressions would have a meaning in terms of the $\xrightarrow{\mathsf{exec}}$ relation. This lets us avoid specifying at what point implementation-level resource limits are hit. The implementation resorts to an error message when Jitawa runs into an arithmetic overflow, attempts to parse a too long symbol (more than 254 characters long), or runs out of room on the heap, stack, symbol table or code heap.

6 The Combination of Milawa and Jitawa

Jitawa runs fast enough and manages its memory well enough to successfully complete the proof of self-verification for Milawa. This is a demanding proof that many Common Lisp systems cannot successfully complete. The full input file is 4 gigabytes and contains 520 million unique conses. On our computer, Clozure Common Lisp—an excellent, state-of-the-art Common Lisp implementation—takes 16 hours to finish the proof; this is with all optimization enabled, garbage collection tuning, and inlining hints that provide significant benefits.

Jitawa is currently about 8x slower for the full proof. While this is considerably slower, it may be adequate for some proof efforts. We are also investigating how performance may be improved. Jitawa is only 20% slower than CCL on the first 4,500 events (about 1.5 hours of computation), so it seems that competitive performance may be possible.

Is there a formal connection between Jitawa and Milawa? We would eventually like to mechanically prove that, when run with Jitawa, Milawa's trusted core is faithful to the Milawa logic. We have not yet done this, but we have at least proved a weaker connection, viz. evaluation in Jitawa respects the 52 axioms [5, Ch. 2] of the Milawa logic that constrain the behavior of Lisp primitives.

For instance, the *Closed Universe Axiom* says every object must satisfy either `natp`, `symbolp`, or `consp`. In Milawa, this is written as:

```
(por* (pequal* (natp x) 't)
      (por* (pequal* (symbolp x) 't)
            (pequal* (consp x) 't)))
```

The corresponding HOL theorem is stated as:

```
valid_sexp ["x"] ("  (por* (pequal* (natp x) 't)           " ++
                  "        (por* (pequal* (symbolp x) 't)  " ++
                  "              (pequal* (consp x) 't)))  ")
```

We are able to copy the axiom statements into HOL nearly verbatim by having valid_sexp use our parser to read the string into its datatype representation.

7 Discussion and Related Work

Theorem provers are generally very trustworthy. The LCF [7] approach has long been used to minimize the amount of code that must be trusted. Harrison [11] has even formally proved—using an altered version of HOL Light—theorems suggesting HOL Light's LCF-style kernel is faithful to its logic. By addressing the correctness of the runtime system used to execute the prover, we further increase our confidence in these systems.

Runtime correctness may be particularly important for theorem provers that employ reflective techniques. In a separate paper [9], Harrison remarks:

> " [...] the final jump from an abstract function inside the logic to a concrete implementation in a serious programming language which *appears to correspond to it* is a glaring leap of faith. "

While we have not proven a formal connection between Milawa and Jitawa, our work suggests it may be possible to verify a theorem prover's soundness down to the concrete machine code which implements it, thus reducing this "glaring leap of faith." We have kept Jitawa's top-level specification (Section 4) as simple as possible to facilitate such a proof.

Most of this paper has dealt with the question: how do we create a verified Lisp system that is usable and scales well? The most closely related work on this topic is the VLISP project [8], which produced a "comprehensively" (not formally) verified Scheme implementation. The subset of Scheme which they address is impressive: it includes strings, destructive updates and I/O. However, their formalisation and proofs did not reach as far as machine or assembly level, as we have done here and in previous work [18].

Recently, Leroy's Coq-verified C compiler [14], which targets PowerPC, ARM and 32-bit x86 assembly, has been extended with new front-ends that makes it compile MiniML [4] and a garbage-collected source language [16]. The latter extension has been connected to intermediate output from the Glasgow Haskell Compiler. Our runtime uses a verified copying garbage collector similar to the sample collector in McCreight et al. [16], but less than 10% of our proof scripts are directly concerned with verification of our garbage collector and interfacing with it; our approach to this is unchanged from our previous paper [18].

Unrelated to Leroy's C compiler, Chlipala [3] has done some interesting verification work, in Coq, on compilation of a functional language: he verified a compiler from a functional language with references and exceptions to a toy assembly language. Chlipala emphasises use of adaptive programs in Coq's tactic language to make proofs robust.

Source code. The code for Jitawa and its 30,000-line verification proof are available at `http://www.cl.cam.ac.uk/~mom22/jitawa/`. Similarly, Milawa is available at `http://www.cs.utexas.edu/~jared/milawa/Web/`.

Acknowledgements. We thank Mike Gordon, Warren Hunt, Scott Owens, Anna Slobadová and Sol Swords for commenting on drafts of this paper. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

1. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook, 2nd edn. Academic Press, London (1997)
2. Boyer, R.S., Hunt Jr., W.A.: Function memoization and unique object representation for ACL2 functions. In: ACL2 2006. ACM, New York (2006)
3. Chlipala, A.J.: A verified compiler for an impure functional language. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL. ACM, New York (2010)
4. Dargaye, Z., Leroy, X.: Mechanized verification of CPS transformations. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 211–225. Springer, Heidelberg (2007)
5. Davis, J.C.: A Self-Verifying Theorem Prover. PhD thesis, University of Texas at Austin (December 2009)
6. Fetzer, J.H.: Program verification: The very idea. Communications of the ACM 31(9), 1048–1063 (1988)
7. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78. Springer, Heidelberg (1979)
8. Guttman, J., Ramsdell, J., Wand, M.: VLISP: A verified implementation of Scheme. Lisp and Symbolic Computation 8(1/2), 5–32 (1995)
9. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Cambridge, UK (1995)
10. Harrison, J.V.: Formalizing basic first order model theory. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 153–170. Springer, Heidelberg (1998)
11. Harrison, J.: Towards self-verification of HOL light. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 177–191. Springer, Heidelberg (2006)
12. 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)
13. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Dordrecht (2000)
14. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) POPL. ACM, New York (2006)
15. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part 1. Communications of the ACM 3(4), 184–195 (1960)
16. McCreight, A., Chevalier, T., Tolmach, A.P.: A certified framework for compiling and executing garbage-collected languages. In: Hudak, P., Weirich, S. (eds.) ICFP. ACM, New York (2010)
17. Myreen, M.O.: Verified just-in-time compiler on x86. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL. ACM, New York (2010)
18. 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)
19. Myreen, M.O., Slind, K., Gordon, M.J.C.: Extensible proof-producing compilation. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 2–16. Springer, Heidelberg (2009)
20. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 294–309. Springer, Heidelberg (2005)
21. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)

Verified Efficient Enumeration of Plane Graphs Modulo Isomorphism

Tobias Nipkow

Institut für Informatik, Technische Universität München

Abstract. Due to a recent revision of Hales's proof of the Kepler Conjecture, the existing verification of the central graph enumeration procedure had to be revised because it now has to cope with more than 10^9 graphs. This resulted in a new and modular design. This paper primarily describes the reusable components of the new design: a while combinator for partial functions, a theory of worklist algorithms, a stepwise implementation of a data type of sets over a quasi-order with the help of tries, and a plane graph isomorphism checker. The verification turned out not to be in vain as it uncovered a bug in Hales's graph enumeration code.

1 Introduction

In 1998, Hales announced the proof of the Kepler Conjecture (about the densest packing of congruent spheres in Euclidean space), which he published in a series of papers, ending with [6]. In addition to the sequence of journal articles, the proof employs three distinct large computations. To remove any doubt about the correctness of the proof due to the unverified computations, Hales started the Flyspeck project (`http://code.google.com/p/flyspeck`) to produce a formal proof of the Kepler Conjecture. An early contribution [17] was the verification of the enumeration of all potential counterexamples (i.e. denser packings) in the form of plane graphs, the so-called *tame* graphs. We confirmed that the so-called *archive* of tame graphs given by Hales is complete (in fact: too large). In a second step, it must be shown that none of these tame graphs constitute a counterexample. Obua [19] verified much of this part of the proof. This paper is about what happened when the geometry underlying the tame graph abstraction changed.

In 2009, Marchal [15] published a new and simpler approach to parts of Hales's proof. As a consequence Hales revised his proof. At the moment, only the online HOL Light theorems and proofs (most of Flyspeck is carried out in HOL Light [8]) reflect this revision (see the Flyspeck web page above). The complete revised proof will appear as a book [7]. Below we call the two versions of the proof the 1998 and the revised version.

The verified enumeration of tame graphs relies on executing functions verified in the theorem prover. Due to the revision, tame graph enumeration ran out of space because of the 10-fold increase in the number of tame graphs and the 100-fold increase in the overall number of graphs that need to be generated.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 281–296, 2011.

Therefore I completely modularized that part of the proof in order to slot in more space-efficient enumeration machinery.

The verification of tame graph enumeration for the 1998 proof [17] was a bit of an anticlimax: at the end we could confirm that Hales's archive of tame graphs (which he had generated with an unverified Java program) was complete. Not so this time: I found two graphs that were missing from Hales's revised archive (which he had generated with a revised version of that Java program). A few days later Hales emailed me:

> *I found the bug in my code! It was in the code that uses symmetry to reduce the search space. This is a bug that goes all the way back to the 1998 proof. It is just a happy coincidence that there were no missed cases in the 1998 proof. This is a good example of the importance of formal proof in computer assisted proofs.*

The main contribution of this paper is an abstract description of the computational tools used in the revised proof, at a level where they can be reused in different settings. In particular, the paper provides the following generic components:

- A while combinator that allows to reason about terminating executions of partial functions without the need for a termination proof.
- Combinators for and beginnings of a theory of worklist algorithms.
- A stepwise implementation of an abstract type of sets over a quasi-order.
- A general schema for stepwise implementation of abstract data types in Isabelle's locale system of modules.
- A simple isomorphism checker for plane graphs.

Much of this paper is not concerned with the formal proof of the Kepler Conjecture per se, and those parts that are, complement [17].

2 Basics

This work is carried out with the Isabelle/HOL proof assistant. HOL conforms largely to everyday mathematical notation. This section summarizes nonstandard notation and features.

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. Sets over type $'a$, type $'a\ set$ (which is just a synonym for $'a \Rightarrow bool$), follow the usual mathematical conventions. The image of a function over a set is denoted by f ` S. Lists over type $'a$, type $'a\ list$, come with the empty list $[]$, the infix constructor $\cdot$, the infix @ that appends two lists, the length function $|.|$ and the conversion function *set* from lists to sets. Names ending in s typically refer to lists. The **datatype** $'a\ option = None \mid Some\ 'a$ is predefined. Implications are displayed either as arrows or as proof rules with horizontal lines.

Locales [2] are Isabelle's version of parameterized theories. A locale is a named context of functions $f_1, \ldots, f_n$ and assumptions $P_1, \ldots, P_m$ about them that is introduced roughly like this:

locale *loc* = **fixes** $f_1 \dots f_n$ **assumes** $P_1 \dots P_m$

The f_i's are the *parameters* of the locale. Every locale implicitly defines a predicate:

$$loc\ f_1 \dots f_n \longleftrightarrow P_1 \wedge \dots \wedge P_m$$

Locales can be hierarchical as in **locale** *loc'* = loc_1 + loc_2 + **fixes** ....

In the context of a locale, definitions can be made and theorems can be proved. This is called the *body* of the locale and can be extended dynamically.

An *interpretation* of a locale

interpretation *loc* $e_1\ \dots\ e_n$

where the e_i's are expressions, generates the proof obligation *loc* $e_1\ \dots\ e_n$ (recall that *loc* is a predicate), and, if the proof obligation is discharged by the user, one obtains all the definitions and theorems from the body of the locale, with each f_i replaced by e_i. For more details see the tutorial on locales [1].

Executability of functions in Isabelle/HOL does not rely on a constructive logic but merely on the ability of the user to phrase a function, by definition or by lemma, as a recursion equation [5]. Additionally we make heavy use of a preprocessor that replaces (by proof!) many kinds of bounded quantifications by suitable list combinators.

3 Worklist Algorithms

3.1 While Combinators

Proving termination of the enumeration of tame graphs is possible but tedious and unneccesary: after all, it does terminate eventually, which is proof enough. But how to define a potentially partial function in HOL? Originally I had solved that problem by brute force with *bounded recursion*: totalize the function with an additional argument of type *nat* that is decreased with every recursive call, return *None* if 0 is reached and *Some r* when the actual result *r* has been reached, and call the function with a sufficiently large initial number. It does the job but is inelegant because unnecessary. What are the alternatives?

Isabelle's standard function definition facility due to Krauss [10] does not require a termination proof, but until termination has been proved, the recursion equations are conditional and hence not executable. This is the opposite of the original while combinator defined in Isabelle/HOL [18]

$$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$$

where $'a$ is the "state" of the computation, the first parameter is the loop test, the second parameter the loop body that transforms the state, and the last parameter is the start state. This combinator obeys the recursion equation

$$while\ b\ c\ s = (\textsf{if}\ b\ s\ \textsf{then}\ while\ b\ c\ (c\ s)\ \textsf{else}\ s)$$

and enables the definition of executable tail-recursive partial functions. But to prove anything useful about the result of the function, we first need to prove termination. This is a consequence of the type of *while*: the result does not tell us if it came out of a terminating computation sequence or is just some arbitrary value forced by the totality of the logic.

Krauss' recent work [11] theoretically provides what we need but the implementation does not yet. Hence Krauss and I defined a while combinator that returns an option value, telling us if it terminated or not, just as in bounded recursion outlined above, but without the need for a counter:

$$\textit{while-option} :: ('a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ \textit{option}$$

$$\begin{array}{l}\textit{while-option}\ b\ c\ s =\\ (\mathsf{if}\ \exists\, k.\ \neg\ b\ (c^k\ s)\ \mathsf{then}\ \textit{Some}\ (c^{\textit{LEAST}\ k.\ \neg\ b\ (c^k\ s)}\ s)\ \mathsf{else}\ \textit{None})\end{array}$$

It obeys a similar unfolding law as *while*

$$\textit{while-option}\ b\ c\ s = (\mathsf{if}\ b\ s\ \mathsf{then}\ \textit{while-option}\ b\ c\ (c\ s)\ \mathsf{else}\ \textit{Some}\ s)$$

but allows to reason about *Some* results in the traditional manner of Hoare logic: invariants hold at the end if they hold at the beginning

$$\frac{\textit{while-option}\ b\ c\ s = \textit{Some}\ t \qquad \forall\, s.\ P\ s \wedge b\ s \longrightarrow P\ (c\ s) \qquad P\ s}{P\ t}$$

and at the end the loop condition no longer holds:

$$\textit{while-option}\ b\ c\ s = \textit{Some}\ t \Longrightarrow \neg\ b\ t$$

Note that termination is built into the premise *while-option* b c s = *Some* t, which is the proposition that we intend to prove by evaluation.

Of course, if we can prove termination by deductive means, this ensures that *Some* result is returned:

$$\frac{\textit{wf}\ \{(t,\ s) \mid (P\ s \wedge b\ s) \wedge t = c\ s\} \qquad \forall\, s.\ P\ s \wedge b\ s \longrightarrow P\ (c\ s) \qquad P\ s}{\exists\, t.\ \textit{while-option}\ b\ c\ s = \textit{Some}\ t}$$

where *wf* R means that relation R is wellfounded, and where P is an invariant.

3.2 Worklist Algorithms

Worklist algorithms repeatedly remove an item from the worklist, replace it by a new list of items, and process the item. They operate on pairs $(ws,\ s)$ of a worklist ws and a state s. We define

$$\begin{array}{l}\textit{worklist-aux}\ \textit{succs}\ f =\\ \textit{while-option}\ (\lambda(ws,\ s).\ ws \neq [])\\ \ (\lambda(ws,\ s).\ \mathsf{case}\ ws\ \mathsf{of}\ x{\cdot}ws' \Rightarrow (\textit{succs}\ s\ x\ @\ ws',\ f\ x\ s))\end{array}$$

of type
$('s \Rightarrow 'a \Rightarrow 'a\ list) \Rightarrow ('a \Rightarrow 's \Rightarrow 's) \Rightarrow 'a\ list \times 's \Rightarrow ('a\ list \times 's)\ option.$
Type $'a$ is the type of items, type $'s$ the type of states. Functions *succs* and f produce the next items and next state. If the algorithm terminates, it must have enumerated the set of items reachable via the successor function starting from the start items.

The successor function may depend on the state. This allows us, for example, to detect loops in the search process by carrying already visited items around in the state. But our application does not require this generality: its successor relationship is a tree. Hence we specialize *worklist-aux* and develop a theory of worklist algorithms on trees. A unified theory of worklist algorithms on trees and graphs is beyond the scope of this paper. From now on *succs* will not depend on the state and we define

$$worklist\text{-}tree\text{-}aux\ succs = worklist\text{-}aux\ (\lambda s.\ succs)$$

Upon termination the worklist will be empty and we project on the state:

$$\begin{array}{l} worklist\text{-}tree\ succs\ f\ ws\ s = \\ (\mathsf{case}\ worklist\text{-}tree\text{-}aux\ succs\ f\ (ws,\ s)\ \mathsf{of}\ None \Rightarrow None \\ \ \mid Some\ (ws,\ s) \Rightarrow Some\ s) \end{array}$$

In order to talk about the set of items reachable via *succs* we introduce the abbreviation

$$Rel\ succs \equiv \{(x,\ y) \mid y \in set\ (succs\ x)\}$$

that translates *succs* into a relation. In addition, R ‘‘ S is the image of a relation over a set. Thus $(Rel\ succs)^*$ ‘‘ A is the set of items reachable from the set of items A via the reflexive transitive closure of *Rel succs*.

The first theorem about *worklist-tree* expresses that it folds f over the reachable items in some order:

$$\frac{worklist\text{-}tree\ succs\ f\ ws\ s = Some\ s'}{\exists\, rs.\ set\ rs = (Rel\ succs)^*\ ``\ set\ ws \wedge s' = fold\ f\ rs\ s}$$

where $fold\ f\ []\ s = s$ and $fold\ f\ (x \cdot xs)\ s = fold\ f\ xs\ (f\ x\ s)$.

This theorem is intentionally weak: *rs* is some list whose elements form the set of reachable items, in any order and with any number of repetitions. The order should not matter, to allow us to replace the particular depth-first traversal strategy of *worklist-aux* by any other, for example appending the successor items at the right end of the worklist. Of course it means that f should also be insensitive to the order. Moreover, f should be insensitive to duplicates, i.e. it should be idempotent. Thus this theorem is specialized for applications where the state is effectively a set of items, which is exactly what we are aiming for in our application, the enumeration and collection of graphs.

If we want to prove some property of the result of *worklist-tree*, we can do so by obtaining *rs* and proving the property by induction over *rs*. But we can also replace the induction by a Hoare-style invariance rule:

$$\frac{\textit{worklist-tree succs f ws s} = \textit{Some } s' \\ \forall s.\ R\ []\ s\ s \qquad \forall r\ x\ ws\ s.\ R\ ws\ (f\ x\ s)\ r \longrightarrow R\ (x{\cdot}ws)\ s\ r}{\exists rs.\ \textit{set rs} = (\textit{Rel succs})^*\ ``\ \textit{set ws} \wedge R\ rs\ s\ s'}$$

This rule is phrased in terms of a predicate R of type $'a\ \textit{list} \Rightarrow {'s} \Rightarrow {'s} \Rightarrow \textit{bool}$, where $R\ rs\ s\ s'$ should express the relationship between some start configuration $(rs,\ s)$ and the corresponding final state s', when the worklist has been emptied.

Unfortunately, this rule is too weak in practice: both the items and the states come with nontrivial invariants of their own, and the invariance of R can only be shown if we may assume that the item and state invariants hold for the arguments of R. In nice set theoretic language the extended rule looks like this:

$$\frac{\textit{worklist-tree succs f ws s} = \textit{Some } s' \qquad \textit{succs} \in I \to \textit{lists } I \\ \textit{set ws} \subseteq I \qquad s \in S \qquad f \in I \to S \to S \qquad \forall s.\ R\ []\ s\ s \\ \forall r\ x\ ws\ s.\ x \in I \wedge \textit{set ws} \subseteq I \wedge s \in S \wedge R\ ws\ (f\ x\ s)\ r \longrightarrow R\ (x{\cdot}ws)\ s\ r}{\exists rs.\ \textit{set rs} = (\textit{Rel succs})^*\ ``\ \textit{set ws} \wedge R\ rs\ s\ s'}$$

Here I and S are the invariants on items and states, *lists* I is the set of lists over I, and $A \to B$ is the set of functions from set A to set B.

As a simple application we obtain almost automatically that the function

$$\textit{colls succs P ws} = \textit{worklist-tree succs}\ (\lambda x\ xs.\ \textsf{if}\ P\ x\ \textsf{then}\ x{\cdot}xs\ \textsf{else}\ xs)\ ws\ []$$

indeed collects all reachable items that satisfy P:

$$\frac{\textit{colls succs P ws} = \textit{Some rs}}{\textit{set rs} = \{x \in (\textit{Rel succs})^*\ ``\ \textit{set ws} \mid P\ x\}}$$

3.3 Sets over a Quasi-Order

In our application we need to collect a large set of graphs and we encounter many isomorphic copies of each graph. Storing all copies is out of the question for reasons of space. Hence we work with sets over a quasi-order, thus generalizing the graph isomorphism to a subsumption relation. We formulate our worklist algorithms in the context of an abstract interface to such a set data type and use Isabelle's locale mechanism (see §2) for this purpose. We start with a locale for the quasi-order, later to be interpreted by graph isomorphism:

locale *quasi-order* =
fixes *qle* :: $'a \Rightarrow {'a} \Rightarrow \textit{bool}$ (**infix** $\preceq$ 60)
assumes $x \preceq x$
and $x \preceq y \Longrightarrow y \preceq z \Longrightarrow x \preceq z$

The following definitions are made in this context:

$x \in_{\preceq} M \equiv \exists y{\in}M.\ x \preceq y$
$M \subseteq_{\preceq} N \equiv \forall x{\in}M.\ x \in_{\preceq} N$
$M =_{\preceq} N \equiv M \subseteq_{\preceq} N \wedge N \subseteq_{\preceq} M$

The actual work will be done in the context of sets over $\preceq$ in locale *set-modulo*, an extension of locale *quasi-order*:

> **locale** $\textit{set-modulo} = \textit{quasi-order} +$
> **fixes** $\textit{empty} :: {}'s$
> **and** $\textit{insert-mod} :: {}'a \Rightarrow {}'s \Rightarrow {}'s$
> **and** $\textit{set-of} :: {}'s \Rightarrow {}'a\ \textit{set}$
> **and** $I :: {}'a\ \textit{set}$
> **and** $S :: {}'s\ \textit{set}$
> **assumes** $\textit{set-of}\ \textit{empty} = \emptyset$
> **and** $x \in I \Longrightarrow s \in S \Longrightarrow \textit{set-of}\ s \subseteq I \Longrightarrow$
> $\quad \textit{set-of}\ (\textit{insert-mod}\ x\ s) = \{x\} \cup \textit{set-of}\ s\ \vee$
> $\quad (\exists\, y{\in}\textit{set-of}\ s.\ x \preceq y) \wedge \textit{set-of}\ (\textit{insert-mod}\ x\ s) = \textit{set-of}\ s$
> **and** $\textit{empty} \in S$
> **and** $s \in S \Longrightarrow \textit{insert-mod}\ x\ s \in S$

In the body of a locale, the type variables in the types of the locale parameters are fixed. Above, $'a$ stands for the fixed element type, $'s$ for the abstract type of sets. The empty set is *empty*, elements are inserted by *insert-mod*. The sets I and S are invariants on elements and sets. In our application later on, both are needed. The behaviour of our sets is specified with the help of an abstraction function *set-of* that maps them back to HOL's mathematical sets.

The first assumption is clear, the last two assumptions state that all sets generated by *empty* and *insert-mod* satisfy the invariant. Only the second assumption needs an explanation, ignoring its self-explanatory preconditions. The point of this assumption is to leave the implementation complete freedom what to do when inserting a new element x. If the set already contains an element y that subsumes x, *insert-mod* is allowed to ignore x. But is not forced to: it may always insert x. This specification allows an implementation to choose how much effort to invest to avoid subsumed elements in a set. Because this subsumption test can be costly: in our application it involves testing for isomorphism with tens of thousands of graphs. Our implementation later on will use a hash function to zoom in on a small subset of potentially isomorphic graphs and only test isomorphism on those. This liberal specification of *insert-mod* saves us from proving that isomorphic graphs have the same hash value.

The above sets only offer *empty* and *insert-mod*, which seems overly toy-like or even useless. In reality there is also a function $\textit{all} :: ({}'a \Rightarrow \textit{bool}) \Rightarrow {}'s \Rightarrow \textit{bool}$ for examining sets, and many other functions could be added to *set-modulo*, but this would not raise any interesting new issues.

3.4 Collecting Modulo Subsumption

This subsection takes place completely within the context of locale *set-modulo*. We specialize the generic worklist combinator to fold *insert-mod* over the list of reachable items. At the same time we parameterize things further: we merely collect those items that satisfy some predicate P, and we don't collect the items themselves but apply some function f to them first:

$$\mathit{insert\text{-}mod2}\ P\ f\ x\ s = (\textsf{if}\ P\ x\ \textsf{then}\ \mathit{insert\text{-}mod}\ (f\ x)\ s\ \textsf{else}\ s)$$

Filtering with P in a separate pass is out of the question in our application because less than one 10^4th of all reachable items satisfy P; trying to store all reachable items would exhaust available memory. Applying f right away, rather than in a second pass, is also done for efficiency but is less critical.

The actual collecting is done by our worklist combinators:

$$\begin{aligned}&\mathit{worklist\text{-}tree\text{-}coll\text{-}aux}\ \mathit{succs}\ P\ f = \mathit{worklist\text{-}tree}\ \mathit{succs}\ (\mathit{insert\text{-}mod2}\ P\ f)\\ &\mathit{worklist\text{-}tree\text{-}coll}\ \mathit{succs}\ P\ f\ \mathit{ws} = \mathit{worklist\text{-}tree\text{-}coll\text{-}aux}\ \mathit{succs}\ P\ f\ \mathit{ws}\ \mathit{empty}\end{aligned}$$

With the help of the generic theorems about *worklist-tree* (see §3.2) and the assumptions of locale *set-modulo* we can derive two important theorems about *worklist-tree-coll*. Its result is equivalent (modulo $\preceq$) to the image under f of those reachable items that satisfy P:

$$\frac{\begin{array}{c}\mathit{worklist\text{-}tree\text{-}coll}\ \mathit{succs}\ P\ f\ \mathit{ws} = \mathit{Some}\ s'\\ \mathit{succs} \in I_0 \to \mathit{lists}\ I_0 \qquad \mathit{set}\ \mathit{ws} \subseteq I_0 \qquad f \in I_0 \to I\end{array}}{\mathit{set\text{-}of}\ s' =_{\preceq} f\ `\ \{x \in (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \mid P\ x\}}$$

This theorem alone leaves the possibility that *set-of* s' contains elements that are only equivalent but not identical to the reachable items. But we can also derive

$$\frac{\begin{array}{c}\mathit{worklist\text{-}tree\text{-}coll}\ \mathit{succs}\ P\ f\ \mathit{ws} = \mathit{Some}\ t\\ \mathit{succs} \in I_0 \to \mathit{lists}\ I_0 \qquad \mathit{set}\ \mathit{ws} \subseteq I_0 \qquad f \in I_0 \to I\end{array}}{\mathit{set\text{-}of}\ t \subseteq f\ `\ \{h \in (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \mid P\ h\}}$$

This is helpful because it means, for example, that the resulting items all satisfy the invariant I.

4 Implementing Sets Modulo

We will now implement the interface *set-modulo* in two steps. First we implement *set-modulo* abstractly by another locale, *set-mod-maps*, with a map-like interface, and then we implement that by a concrete data structure, *Tries*. Figure 1 depicts the implementation relationships (where $A \to B$ means that A is implemented by B), and $\supseteq$ is locale extension. The full meaning of the diagram will become clear as we go along.

$$\begin{tikzcd}\mathit{set\text{-}modulo} \arrow[r] & \mathit{set\text{-}mod\text{-}maps} \arrow[r, phantom, "\supseteq"] \arrow[d] & \mathit{maps} \arrow[d]\\ & \mathit{Graph} & \mathit{Tries}\end{tikzcd}$$

Fig. 1. Implementation diagram

4.1 Maps

Our maps correspond to functions of type *'a* ⇒ *'b list* that return [] almost everywhere. We could implement them via ordinary maps of type *'a* ⇒ *'c option* as they are provided, for example, in the Collections Framework [12]. The latter did not exist yet when our proof was first developed, and since *'a* ⇒ *'b list* is simpler than *'a* ⇒ *'b list option* (where *None* acts like []) and of interest in its own right, this is what locale *maps* specifies:

locale *maps* =
fixes *empty* :: *'m*
and *up* :: *'m* ⇒ *'a* ⇒ *'b list* ⇒ *'m*
and *map-of* :: *'m* ⇒ *'a* ⇒ *'b list*
and *M* :: *'m set*
assumes *map-of empty* = $(\lambda a.\ [])$
and *map-of* (*up m a bs*) = *fun-upd* (*map-of m*) *a bs*
and *empty* ∈ *M*
and *m* ∈ *M* ⟹ *up m a bs* ∈ *M*

Type variable *'m* represents the maps, *'a* and *'b list* its domain and range type. Maps are created from *empty* by *up* (update). Function *map-of* serves two purposes: as a lookup function and as an abstraction function from *'m* to *'a* ⇒ *'b list*. Function *fun-upd* is the predefined pointwise function update.

We extend *maps* with a function that produces the set of elements in the range of a map:

set-of m = $(\bigcup_x$ *set* (*map-of m x*)$)$

4.2 Implementing Sets Modulo by Maps

Before we present the details, we sketch the rough idea. Sets of elements of type *'b* are represented by maps from *'a* to *'b list* where *'a* is some type of "addresses" and there is some (hash) function *key* :: *'b* ⇒ *'a*. Operation *insert-mod x m* will operate as follows: it looks up *key x* in *m*, obtains some list *ys*, checks if *x* is subsumed (modulo $\preceq$) by some element of *ys*, and adds it otherwise.

This abstract implementation is phrased as a locale that enriches *maps* with *key* and some further functions, and that will implement the operations of *set-modulo* with the help of those of *maps* and the newly added functions.

locale *set-mod-maps* = *maps* + *quasi-order* +
fixes *subsumed* :: *'b* ⇒ *'b* ⇒ *bool*
and *key* :: *'b* ⇒ *'a*
and *I* :: *'b set*
assumes $x{\in}I \Longrightarrow y{\in}I \Longrightarrow$ *subsumed x y* $\longleftrightarrow (x \preceq y)$

The two parameters of *set-mod-maps* in addition to *key* are a predicate *subsumed*, meant to represent an executable version of the mathematical $\preceq$, and an invariant *I* that guarantees that *subsumed* coincides with $\preceq$ (see the assumption).

In the body of *set-mod-maps* the actual implementation of *insert-mod* is defined:

$$\begin{array}{l}
\textit{insert-mod}\ x\ m = \\
(\mathsf{let}\ k = \textit{key}\ x;\ \textit{ys} = \textit{map-of}\ m\ k \\
\ \mathsf{in\ if}\ \exists\, y{\in}\textit{set}\ \textit{ys}.\ \textit{subsumed}\ x\ y\ \mathsf{then}\ m\ \mathsf{else}\ \textit{up}\ m\ k\ (x{\cdot}\textit{ys}))
\end{array}$$

Now it is time to assert and prove that *set-mod-maps* is an implementation of *sets-modulo*, i.e. that we can interpret *sets-modulo* in the context of *set-mod-maps*:[1]

$$\begin{array}{l}
\textbf{interpretation}\ (\textbf{in}\ \textit{set-mod-maps}) \\
\quad \textit{set-modulo}\ (\textit{op}\ \preceq)\ \textit{empty}\ \textit{insert-mod}\ \textit{set-of}\ I\ M
\end{array}$$

Predicate *set-modulo* takes six arguments. The first one is the quasi-order from locale *quasi-order* that it extends. Since *set-mod-maps* also extends *quasi-order*, we can supply that same relation $\preceq$. The other five parameters are the ones fixed in *set-modulo*: the empty set is interpreted by the empty map, *insert-mod* is interpreted by the *insert-mod* defined just above, *set-of* is interpreted by the *set-of* defined in the context of *maps*, the invariant on set elements is interpreted by the parameter *I*, and the set invariant by the maps invariant. The proofs of the *set-modulo* assumptions under this interpretation are straightforward.

Thus we have realized the horizontal arrow in Figure 1: any implementation (i.e. interpretation) of *set-mod-maps* yields an implementation of *set-modulo*.

4.3 Implementing Maps by Tries

Tries (pronounced as in "retrieval") are search trees indexed by lists of keys, one element for each level of the tree. Ordinary tries are found in the Collections Framework [12]; we provide a simple variation aimed at *maps*: lists rather than single items are stored, obviating the need for options.

Tries are defined in theory *Tries* and we refer to many of its operations with their qualified name, i.e. *Tries.f* rather than just *f*. The datatype itself is defined like this, where *'a* is the type of keys and *'v* the type of values:

$$\textbf{datatype}\ ('a,\ 'v)\ \textit{tries} = \textit{Tries}\ ('v\ \textit{list})\ (('a \times ('a,\ 'v)\ \textit{tries})\ \textit{list})$$

The two projection functions are *values* (*Tries vs al*) = *vs* and *alist* (*Tries vs al*) = *al*. The name *alist* reflects that the second argument is an association list of keys and subtries. The invariant *Tries.inv* asserts that there are no two distinct elements with the same key in any alist in some trie. For concreteness, here is the code for *lookup* and *update*:

$$\begin{array}{l}
\textit{Tries.lookup}\ t\ [] = \textit{values}\ t \\
\textit{Tries.lookup}\ t\ (a{\cdot}\textit{as}) = \\
(\mathsf{case}\ \textit{map-of}\ (\textit{alist}\ t)\ a\ \mathsf{of}\ \textit{None} \Rightarrow []\ |\ \textit{Some}\ \textit{at} \Rightarrow \textit{Tries.lookup}\ \textit{at}\ \textit{as})
\end{array}$$

[1] The actual syntax uses the keyword **sublocale** but we have chosen this more intuitive variation of **interpretation**.

$$\begin{array}{l}
\mathit{Tries.update}\ t\ []\ \mathit{vs} = \mathit{Tries}\ \mathit{vs}\ (\mathit{alist}\ t)\\
\mathit{Tries.update}\ t\ (a{\cdot}\mathit{as})\ \mathit{vs} =\\
(\mathsf{let}\ \mathit{tt} = \mathsf{case}\ \mathit{map\text{-}of}\ (\mathit{alist}\ t)\ a\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{Tries}\ []\ []\ |\ \mathit{Some}\ \mathit{at} \Rightarrow \mathit{at}\\
\ \mathsf{in}\ \mathit{Tries}\ (\mathit{values}\ t)\ ((a,\ \mathit{Tries.update}\ \mathit{tt}\ \mathit{as}\ \mathit{vs}){\cdot}\mathit{rem\text{-}alist}\ a\ (\mathit{alist}\ t)))
\end{array}$$

Auxiliary functions are omitted and easy to reconstruct. Now it is straightforward to show

interpretation *maps* (*Tries* [] []) *Tries.update Tries.lookup Tries.inv*

Thus we have realized the arrow from *maps* to *Tries* in Figure 1. Once we also implement the *set-mod-map* extension (in §5.2), we obtain the body of *set-modulo*, in particular the collecting worklist algorithms and their correctness theorems.

4.4 Stepwise Implementation in General

The above developments are instances of the following general schema, simplified for the sake of presentation. We want to implement an abstract interface

locale A = **fixes** f **assumes** P

In our case A is *set-modulo.* We base the implementation on n interfaces

locale B_i = **fixes** g_i **assumes** Q_i $\qquad (i = 1, \ldots, n)$

In our case, there are two B_i's: *maps*, and the extension of *set-mod-maps* with *key*, *subsumed* and *I*, which can be viewed as a separate locale that is added to the import list of *set-mod-maps.* Now, given some schema $F[g_1, \ldots, g_n]$ for defining f from the $g_1, \ldots, g_n$, A can be implemented by the B_i's:

locale *A-by-Bs* = $B_1 + \ldots + B_n$ + **definition** *fimpl* = $F[g_1, \ldots, g_n]$

The correctness of this development step corresponds to the claim that A can be interpreted in the context of *A-by-Bs*, which of course needs to be proved:

interpretation (**in** *A-by-Bs*) *A fimpl*

Each B_i can either be implemented in the same manner as A, or it can be implemented directly:

interpretation B_i *concrete-*g_i

where *concrete-*g_i in an implementation of g_i on a suitable concrete type. In the end, we obtain an overall concrete implementation of A, together with an instance of the body of A.

It seems that this is the first time a general development scheme for abstract data types has been formulated for locales. The general idea of stepwise development of abstract data types via theory interpretations goes back to Maibaum *et al.* [14]. Theory interpretations are also a central concept in IMPS [3], but with a focus on mathematics. Likewise, locales have primarily been motivated as a device for structuring mathematics [9]. Instances of the above schema can be found in a few large Isabelle developments, for example Lochbihler's Java-like language with threads [13], but even the Collections Framework by Lammich and Lochbihler [12], which is all about abstract data type specification and implementation, does not discuss the general picture and does not contain an instance of *A-by-Bs* above.

Similar specifications and developments are possible with the Coq module system, e.g. [4]. It would be interesting to investigate the precise relationship between locales and the Coq module system.

5 Application to Plane Graphs

As explained in the Introduction, Hales's proof involves the enumeration of a very large set of tame graphs, where tame graphs are by definition also plane [17]. Our representation of plane graphs follows Hales's 1998 proof: a plane graph is a set/list of faces, where each face is a list of vertices of type $'a$:

$$'a\ \mathit{Fgraph} = {}'a\ \mathit{list\ set}$$
$$'a\ \mathit{fgraph} = {}'a\ \mathit{list\ list}$$

Type *Fgraph* involves sets and belongs to the mathematical level, type *fgraph* represents sets by lists and belongs to the executable level. Below we develop a number of notions first on the *Fgraph* level and transfer them easily and directly to the *fgraph* level. We call graphs on both levels *face graphs* and frequently use the letter F for faces.

5.1 Plane Graph Isomorphisms

This subsection describes in some detail the isomorphism test that had to be left out of [17].

Face graphs need to be compared modulo rotation of faces and we define

$$F_1 \cong F_2 \equiv \exists\, n.\ F_2 = \mathit{rotate}\ n\ F_1$$
$$\{\cong\} \equiv \{(F_1,\ F_2) \mid F_1 \cong F_2\}$$

Relation $\{\cong\}$ is an equivalence and we can form the quotient $S\ //\ \{\cong\}$ with the predefined quotient operator $//$, defining proper homomorphisms and isomorphisms on face graphs, where φ is a function on vertices:

$$\mathit{is\text{-}pr\text{-}Hom}\ \varphi\ \mathit{Fs}_1\ \mathit{Fs}_2 \equiv \mathit{map}\ \varphi\ `\ \mathit{Fs}_1\ //\ \{\cong\} = \mathit{Fs}_2\ //\ \{\cong\}$$
$$\mathit{is\text{-}pr\text{-}Iso}\ \varphi\ \mathit{Fs}_1\ \mathit{Fs}_2 \equiv \mathit{is\text{-}pr\text{-}Hom}\ \varphi\ \mathit{Fs}_1\ \mathit{Fs}_2 \wedge \mathit{inj\text{-}on}\ \varphi\ (\bigcup_{F \in \mathit{Fs}_1}\ \mathit{set}\ F)$$
$$\mathit{is\text{-}pr\text{-}iso}\ \varphi\ \mathit{Fs}_1\ \mathit{Fs}_2 \equiv \mathit{is\text{-}pr\text{-}Iso}\ \varphi\ (\mathit{set}\ \mathit{Fs}_1)\ (\mathit{set}\ \mathit{Fs}_2)$$

The first two functions operate on type *Fgraph*, the last one on *fgraph*. The attribute "proper" indicates that orientation of faces matters. The more liberal version where the faces of one graph may have the reverse orientation of those of the other graph, which corresponds to the standard notion of graph isomorphism, is easily defined on top:

$$\begin{aligned}
&\textit{is-Iso}\ \varphi\ Fs_1\ Fs_2 \equiv \textit{is-pr-Iso}\ \varphi\ Fs_1\ Fs_2 \lor \textit{is-pr-Iso}\ \varphi\ Fs_1\ (\textit{rev}\ `\ Fs_2)\\
&\textit{is-iso}\ \varphi\ Fs_1\ Fs_2 \equiv \textit{is-Iso}\ \varphi\ (\textit{set}\ Fs_1)\ (\textit{set}\ Fs_2)\\
&g_1 \simeq g_2 \equiv \exists\,\varphi.\ \textit{is-iso}\ \varphi\ g_1\ g_2
\end{aligned}$$

What we need is an executable isomorphism test. A simple solution would have been to search for an isomorphism by some unverified function and check the result with a verified checker. Although this is a perfectly reasonable solution, we wanted to see if a verified isomorphism test that performs well in our context is also within easy reach. It turns out it is. The verification took of the order of 600 lines of proof that rely heavily on automation of set theory.

We start with the search for a proper isomorphism. The isomorphism is represented by a list of vertex pairs I that is built up incrementally. Given two lists of faces, repeatedly take a face F_1 from the first list, find a matching face F_2 in the second list, and remove both faces. Matching means that F_1 and F_2 must have the same length, and for some $n < |F_2|$, the bijection obtained by pairing off F_1 and *rotate* n F_2 vertex by vertex is compatible with I. Then we can merge it with I. This is the corresponding recursive function:

$$\textit{pr-iso-test-rec} :: ('a \times 'b)\textit{list} \Rightarrow 'a\ \textit{fgraph} \Rightarrow 'b\ \textit{fgraph} \Rightarrow \textit{bool}$$

```
$pr\text{-}iso\text{-}test\text{-}rec\ I\ []\ Fs_2 \longleftrightarrow Fs_2 = []$
$pr\text{-}iso\text{-}test\text{-}rec\ I\ (F_1 \cdot Fs_1)\ Fs_2 \longleftrightarrow$
$(\exists\,F_2 \in set\ Fs_2.$
    $|F_1| = |F_2| \land$
    $(\exists\,n < |F_2|.$
        let $I' = zip\ F_1\ (rotate\ n\ F_2)$
        in $compat\ I'\ I \land$
           $pr\text{-}iso\text{-}test\text{-}rec\ (merge\ I'\ I)\ Fs_1\ (remove1\ F_2\ Fs_2)))$
```

Function *compat* checks if two lists of vertex pairs are compatible

$$\textit{compat}\ I\ I' \equiv \forall\,(x,\ y)\in \textit{set}\ I.\ \forall\,(x',\ y')\in\textit{set}\ I'.\ x = x' \longleftrightarrow y = y'$$

and function *merge* merges them:

```
$merge\ []\ I = I$
$merge\ (xy \cdot xys)\ I =$
$($let $(x,\ y) = xy$
 in if $\forall\,(x',\ y') \in set\ I.\ x \neq x'$ then $xy \cdot merge\ xys\ I$ else $merge\ xys\ I)$
```

Moving from proper isomorphism to isomorphism is easy

$$\textit{iso-test}\ g_1\ g_2 \longleftrightarrow \textit{pr-iso-test}\ g_1\ g_2 \lor \textit{pr-iso-test}\ g_1\ (\textit{map rev}\ g_2)$$

where *pr-iso-test* Fs_1 $Fs_2 \longleftrightarrow$ *pr-iso-test-rec* $[]$ Fs_1 Fs_2.

Function *pr-iso-test-rec* is the result of a stepwise development that we skip in favour of the final correctness theorem:

$$\textit{pr-iso-test-rec}\ []\ Fs_1\ Fs_2 \longleftrightarrow (\exists\,\varphi.\ \textit{is-pr-iso}\ \varphi\ Fs_1\ Fs_2)$$

This theorem comes with a number of preconditions on the two face lists: in each face, all vertices must be distinct, all faces in each list must be distinct modulo $\cong$, and the empty face is not allowed. An executable version of these preconditions, for the case where the vertices are natural numbers, can be expressed like this:

$$\begin{aligned}&\textit{pre-iso-test Fs} \longleftrightarrow\\ &[] \notin \textit{set Fs} \wedge (\forall F{\in}\textit{set Fs. distinct } F) \wedge \textit{distinct } (\textit{map rotate-min Fs})\end{aligned}$$

Function *rotate-min* produces a unique representative of the $\cong$ equivalence class of a face by rotating the minimal vertex to the head of the list.

The key theorem now states that under the executable preconditions, the executable and the mathematical definition of isomorphism agree:

$$\textit{pre-iso-test } Fs_1 \Longrightarrow \textit{pre-iso-test } Fs_2 \Longrightarrow \textit{iso-test } Fs_1\ Fs_2 \longleftrightarrow Fs_1 \simeq Fs_2$$

5.2 Sets of Graphs Modulo Isomorphism

Now we can reap the benefits of the implementation work in §3.4. The interpretation of locale *quasi-order* with $\simeq$ on type *fgraph* is trivial and omitted. The interpretation of *set-mod-maps* is more involved:

interpretation
set-mod-maps (*Tries* [] []) *Tries.update Tries.lookup Tries.inv*
(*op* $\simeq$) *iso-test hash pre-iso-test*

The first four parameters are identical to those in the interpretation of the sublocale *maps* in §4.2. The quasi-order is interpreted as $\simeq$. The last three parameters interpret the *subsumed*, *key* and *I* parameters of *set-mod-maps*. Only the hash function remains to be explained, informally. It takes an *fgraph* and produces a list of natural numbers, in this order: the number of vertices, the number of faces, the sorted list of degrees of each vertex. All of these are well-known invariants under isomorphism, but as explained in §3.3, we have set things up such that we do not need to prove this.

The final theorem in the previous subsection is all we need to prove the one assumption of locale *set-mod-maps* in this interpretation. Now we have established the last remaining arrow in Figure 1, the one from to *set-mod-maps* to *Graph* (representing the plane graph theory).

5.3 Application to Hales's Proof

During the enumeration, graphs are represented by an abstract type *graph* with a successor function *next* :: *graph* $\Rightarrow$ *graph list*, a predicate *final* that picks out the tame graphs, and a projection *fgraph* :: *graph* $\Rightarrow$ *nat fgraph*.

The interpretation of *set-mod-maps* above yields a function *worklist-tree-coll* that we can specialize as follows to the enumeration of all tame graphs:

worklist-tree-coll next final fgraph

of type *graph list* ⇒ (*nat*,*nat fgraph*) *tries option*. In the end, this function is applied to four different start graphs, runs for 11 hours, and terminates with *Some* tries; the resulting tries are compared modulo isomorphism with an archive of tame graphs, which Hales had initially supplied. The first time the verified enumeration terminated successfully, I found that the archive lacked two graphs. The completed archive is available online, as are all the Isabelle theories [16].

During the enumeration, a total of 1 870 507 512 graphs are generated, of which 348 231 are final tame graphs, of which 18 762 are distinct modulo isomorphism. The final tame graphs have at most 25 faces (18.6 on average) and at most 15 vertices (13.8 on average). Our hash function works very well: on average, there are 3.1 graphs in each entry of a trie, and in the worst case there are 97.

6 Conclusion

This work is an encouraging example of both the contribution that theorem proving can make to extreme mathematical proofs and the contribution that software development methods can make to theorem proving. Initially we had hacked our way through the proof and did not describe the details in [17]. This paper is a rational reconstruction of the underlying data structures and algorithms of that hack. This exercise in modularization has given rise to a number of interesting reusable components.

Acknowledgement. I would like to thank Tom Hales for hosting my visit to Pitt, Jasmin Blanchette for grammatical and stylistic scrutiny, and Alex Krauss for the subsumption relation and many other improvements.

References

1. Ballarin, C.: Tutorial to locales and locale interpretation, `http://isabelle.in.tum.de/dist/Isabelle2011/doc/locales.pdf`
2. 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)
3. Farmer, W.: Theory interpretation in simple type theory. In: Heering, J., Meinke, K., Möller, B., Nipkow, T. (eds.) HOA 1993. LNCS, vol. 816, pp. 96–123. Springer, Heidelberg (1994)
4. Filliâtre, J.-C., Letouzey, P.: Functors for Proofs and Programs. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 370–384. Springer, Heidelberg (2004)
5. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
6. Hales, T.C.: Sphere packings, VI. Tame graphs and linear programs. Discrete and Computational Geometry 36, 205–265 (2006)
7. Hales, T.C.: Dense sphere packings: A blueprint for formal proofs. Cambridge University Press, Cambridge (forthcoming)

8. Harrison, J.: HOL light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009)
9. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A sectioning concept for isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 149–165. Springer, Heidelberg (1999)
10. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Automated Reasoning 44, 303–336 (2010)
11. Krauss, A.: Recursive definitions of monadic functions. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) Workshop on Partiality and Recursion in Interactive Theorem Proving (PAR 2010). EPTCS, vol. 43, pp. 1–13 (2010)
12. 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)
13. Lochbihler, A.: Jinja with threads. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (2007) Formal proof development, `http://afp.sourceforge.net/entries/JinjaThreads.shtml`
14. Maibaum, T.S.E., Veloso, P.A.S., Sadler, M.R.: A theory of abstract data types for program development: Bridging the gap? In: Ehrig, H., Floyd, C., Nivat, M., Thatcher, J. (eds.) TAPSOFT 1985. LNCS, vol. 186, pp. 214–230. Springer, Heidelberg (1985)
15. Marchal, C.: Study of the Kepler's conjecture: the problem of the closest packing. Mathematische Zeitschrift (2009) (published online)
16. Nipkow, T.: Flyspeck I: Tame graphs. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (2011) Formal proof development, `http://afp.sf.net/devel-entries/Flyspeck-Tame.shtml`
17. 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)
18. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
19. Obua, S., Nipkow, T.: Flyspeck II: The basic linear programs. Annals of Mathematics and Artificial Intelligence 56, 245–272 (2009)

Mechanised Computability Theory

Michael Norrish

[1] Canberra Research Lab., NICTA
[2] Australian National University

Abstract. This paper presents a mechanisation of some basic computability theory. The mechanisation uses two models: the recursive functions and the λ-calculus, and shows that they have equivalent computational power. Results proved include the Recursion Theorem, an instance of the *s-m-n* theorem, the existence of a universal machine, Rice's Theorem, and closure facts about the recursive and recursively enumerable sets. The mechanisation was performed in the HOL4 system and is available online.

1 Introduction

This paper describes mechanisation work in one of computer science's foundational areas: computability theory. This is the theory of what can and cannot be computed by abstract computing machines, using models such as Turing machines, register machines, the λ-calculus and the recursive functions. This paper's focus is on the last two of these models, mainly because of their simplicity (in the case of the recursive functions), and because an existing background theory was available (in the case of the λ-calculus).

By showing the computational equivalence of the two models, we gain additional assurance that their mechanisations are correct. The other standard results, showing what the models are and are not capable of, further validate the work.

Mechanisation in an area such as this is intellectually satisfying in itself. Additionally, the development should provide the wherewithal to mechanise computability arguments where this has not been possible before. For example, Urban, Cheney and Berghofer's impressive paper, *Mechanising the Metatheory of LF* [10] includes an argument to the effect that the algorithm they have formalised (and shown correct) is indeed computable. In the absence of a theory of computability, the argument that the rules of the algorithm are computable is by a combination of careful discussion, suggestive theorems, and (necessarily un-formalised) human inspection.

Contributions

- The first mechanisation of the λ-calculus as a model of computation, including standard auxiliary notions such as the Church numerals.
- A mechanised proof of computational equivalence between two different models: the λ-calculus and the recursive functions.
- Mechanised proofs of a number of standard results from computability theory.
- Discussion of the results and theorem-proving techniques that made the above possible, including use of: the isomorphism between de Bruijn terms and quotiented λ-terms, the standardisation theorem, simplification with pre-orders, and bracket abstraction.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 297–311, 2011.

HOL4 Notation and Theorems All statements appearing with a turnstile ($\vdash$), or as natural deduction style rules, are HOL4 theorems, automatically pretty-printed to LaTeX from the relevant theory in the HOL4 development. Notation specific to this paper is explained as it is introduced. Otherwise, HOL4 supports a notation that is a generally pleasant combination of quantifiers ($\forall$, $\exists$) and functional programming (λ for function abstraction, juxtaposition for function application). Hilbert choice is available with the syntax $(\varepsilon x.\ P\ x)$, meaning "the x such that P holds". Such a term has an unspecified value if there is no such x.

The paper also uses the polymorphic option type (α `option`), with possible values $\mathsf{SOME}\ x$ and NONE. The THE function maps $\mathsf{SOME}\ x$ to x, and is unspecified on NONE. The term $\mathsf{OPTION\_MAP}\ f\ x$ returns $\mathsf{SOME}\ (f\ y)$ when x is $\mathsf{SOME}\ y$, and NONE when x is NONE.

Lists are constructed with the infix "cons" function ::. The length of a list ℓ is written $|\ell|$. Lists support other standard operations such as MAP.

Availability. The sources for the mechanisation described in this paper are available as part of the standard HOL4 distribution ("Kananaskis-6" release), available from `hol.sourceforge.net`.

2 The λ-Calculus: First Steps

This work would not have been possible without earlier mechanisation effort targetting relevant aspects of the λ-calculus. In particular, it relies on my earlier proof of the standardisation theorem [3], and Vestergaard's and my proof [4] that the de Bruijn terms and their associated notion of β-reduction are isomorphic to the λ-terms (quotiented name-carrying syntax) with *their* own notion of β-reduction.

The λ-terms used in this earlier work, and thus in this paper also, are either variables ($\underline{v}$), (left-associating) applications ($M \diamond N$) or abstractions $(\boldsymbol{\lambda} v.\ M)$. These are terms of the object language: the bold lambda is a constructor (which takes a string and a λ-term as arguments) creating a value of type `term` within the higher-order logic. In contrast, the normal lambda of the meta-language creates values in the logic's function spaces. Similarly, the variable constructor, denoted with underlining, injects values from the HOL type of strings into the term type.

These terms are quotiented, and so have equality results such as

$$(\boldsymbol{\lambda} v.\ \underline{v}) = (\boldsymbol{\lambda} u.\ \underline{u})$$

The free variable function over terms is written FV, and the substitution notation is $M[v := N]$, meaning the term resulting from substituting term N for the (free) variable with name v throughout term M.

2.1 Normal Order Reduction

Definition 1. *To guarantee that λ-evaluations find normal forms, we use normal order reduction:*

$$\frac{}{(\lambda v.\ M) \diamond N \longrightarrow_n M[v := N]}$$

$$\frac{M_1 \longrightarrow_n M_2}{(\lambda v.\ M_1) \longrightarrow_n (\lambda v.\ M_2)}$$

$$\frac{M_1 \longrightarrow_n M_2 \quad \neg\mathtt{is\_abs}\ M_1}{M_1 \diamond N \longrightarrow_n M_2 \diamond N}$$

where the predicate `is_abs` *is true of a term if it is an abstraction.*

We are then able to prove that if a term can β-reduce to a β-normal form, then a (necessarily deterministic) normal reduction will eventually arrive at the same place:

$$\vdash M \longrightarrow_\beta^* N \wedge \mathtt{bnf}\ N \Rightarrow M \longrightarrow_n^* N$$

The proof is as per Barendregt [1, §13.2]: in essence, a standard reduction (in the sense of the standardisation theorem) that reaches a β-normal form must also be normal-order as such a reduction can't have ignored a potential redex in its sweep across a term (outermost, left-to-right). By the standardisation theorem, all β-reductions can be emulated by a standard reduction, and so all β-reductions to normal forms can be emulated by normal order reduction.

2.2 Rewriting with β-Equivalence; Bracket Abstraction

In developing the λ-calculus implementations of types such as numbers and de Bruijn terms, it is critical to be able to prove facts of the form $M \equiv_\beta M'$, stating that M is β-equivalent to M'. (The β-equivalence relation is the symmetric, reflexive, transitive closure of the relation that reduces one β-redex.)

The HOL4 simplifier supports rewriting with arbitrary pre-orders, and rewriting with an equivalence (where we additionally have symmetry) is generally quite pleasant. One has to provide introduction rules such as

$$\frac{M_1 \equiv_\beta M_2 \quad N_1 \equiv_\beta N_2}{M_1 \equiv_\beta N_1 \iff M_2 \equiv_\beta N_2}$$

which switches the simplifier from rewriting an equality (boolean equivalence in this case) to β-equivalence. In addition, one can use the following rewrites

$$\vdash \mathtt{bnf}\ N \Rightarrow (M \longrightarrow_n^* N \iff M \longrightarrow_\beta^* N)$$
$$\vdash \mathtt{bnf}\ N \Rightarrow (M \longrightarrow_\beta^* N \iff M \equiv_\beta N)$$

to move to rewriting with $\equiv_\beta$ from goals mentioning $\longrightarrow_n^*$ and $\longrightarrow_{fi}^*$.

It's very important to be able to rewrite with theorems already proved, results such as (see Section 2.3 below for more on Church numerals and arithmetic)

$$\vdash \mathtt{cplus} \diamond \mathtt{church}\ m \diamond \mathtt{church}\ n \longrightarrow_n^* \mathtt{church}\ (m + n)$$

This theorem is a statement about normal order reduction, not β-equivalence, but the simplifier is primed by the inclusion theorem:

$$\vdash M \rightarrow_n^* N \Rightarrow M \equiv_\beta N$$

and is able to use the above as a rewrite. Because β-equivalence is a congruence, such rewrites can be applied at any point within a term.

The basic rule governing β-redexes is present too:

$$\frac{}{(\lambda x.\ M) \diamond N \equiv_\beta M[x := N]}$$

but use of this rule is best avoided because of the possibility that bound variables will need to be renamed.

One might initially hope not to have to deal with variable renaming in a setting where the terms are already quotiented with respect to α-equivalence. Indeed, there is no *semantic* problem, but rather a pragmatic problem to do with making simplification as smooth as possible. The problem stems from the abstraction clause of the substitution rewrite:

$$\vdash v \neq u \wedge v \notin \mathtt{FV}\ N \Rightarrow (\lambda v.\ M)[u := N] = (\lambda v.\ M[u := N])$$

It is not necessarily the case that the bound v will always be fresh with respect to the particular N. In that situation, the desired rewrite could be made to go through by first proving

$$(\lambda v.\ M) = (\lambda w.\ M[v := \underline{w}])$$

where w was chosen to be suitably fresh. Then this equality could be used to substitute "equals-for-equals", and the simplifier would end up simplifying the term

$$M[v := \underline{w}][u := N]$$

before proceeding further.

Implementing this is certainly possible, but would involve writing special-purpose code in ML that the simplifier could call out to as it traversed a term. It seems cleaner to use a technique that can work with the simplifier "as is". Our approach is a procedure inspired by bracket abstraction. The core theorems are shown in Figure 1. These can be applied automatically by the simplifier to prove β-equivalence results between terms with abstractions and terms without.

For example, the original definition of addition on Church numerals is[1]

$$\vdash \mathtt{cplus} = (\lambda\mathtt{"m"}\ \mathtt{"n"}.\ \underline{\mathtt{"m"}} \diamond \underline{\mathtt{"n"}} \diamond \mathtt{csuc})$$

but the application of the rewrites above returns the point-free characterisation:

$$\vdash \mathtt{cplus} \equiv_\beta \mathtt{C} \diamond (\mathtt{B} \diamond \mathtt{C} \diamond (\mathtt{C} \diamond \mathtt{B} \diamond \mathtt{I})) \diamond \mathtt{csuc}$$

The combinator terms B, C, I, K and S are defined as abstractions, but these definitions are not unfolded by the simplifier. Instead, the combinatory characterisations are used as rewrites:

[1] Note how we have to pick concrete names, "x" and "y", for the variables that are "bound" at the object-level. Though $x \neq y \Rightarrow \mathtt{cplus} = (\lambda x\ y.\ \underline{x} \diamond \underline{y} \diamond \mathtt{csuc})$ is true, any attempt to define cplus this way would stumble on the requirement to keep x and y apart, and on the fact that the definition would have (from HOL's perspective) free variables (x and y) on its RHS.

$$\begin{array}{l}
\vdash v \notin \mathtt{FV}\ M \land v \in \mathtt{FV}\ N \Rightarrow (\lambda v.\ M \diamond N) \equiv_\beta \mathtt{B} \diamond M \diamond (\lambda v.\ N) \\
\vdash (\lambda v.\ \mathtt{B} \diamond \underline{v}) \equiv_\beta \mathtt{B} \\
\vdash v \in \mathtt{FV}\ M \land v \notin \mathtt{FV}\ N \Rightarrow (\lambda v.\ M \diamond N) \equiv_\beta \mathtt{C} \diamond (\lambda v.\ M) \diamond N \\
\vdash v \notin \mathtt{FV}\ M \Rightarrow (\lambda v.\ M) \equiv_\beta \mathtt{K} \diamond M \\
\vdash v \in \mathtt{FV}\ M \land v \in \mathtt{FV}\ N \Rightarrow (\lambda v.\ M \diamond N) \equiv_\beta \mathtt{S} \diamond (\lambda v.\ M) \diamond (\lambda v.\ N) \\
\vdash (\lambda x.\ \underline{x}) \ \mathtt{=}\ \mathtt{I}
\end{array}$$

Fig. 1. β-Equivalence rewrites implementing bracket abstraction. As this is β-equivalence, rather than $\beta\eta$-equivalence, we cannot η-contract freely. However, some η-contractions (as in the second rewrite above) are β-valid because the other half of the body is really an abstraction.

$$\begin{array}{l}
\vdash \mathtt{B} \diamond f \diamond g \diamond x \equiv_\beta f \diamond (g \diamond x) \\
\vdash \mathtt{C} \diamond f \diamond x \diamond y \equiv_\beta f \diamond y \diamond x \\
\vdash \mathtt{I} \diamond x \equiv_\beta x \\
\vdash \mathtt{K} \diamond x \diamond y \equiv_\beta x \\
\vdash \mathtt{S} \diamond f \diamond g \diamond x \equiv_\beta f \diamond x \diamond (g \diamond x)
\end{array}$$

2.3 Church Arithmetic

Definition 2. *It is straightforward to encode numbers and other algebraic types within the λ-calculus using the method due to Church. For example, the natural numbers can be injected with the function* `church`*, of type* $\mathtt{num} \longrightarrow \mathtt{term}$*, which is defined:*

$$\vdash \mathtt{church}\ n\ \mathtt{=}\ (\lambda\mathtt{"z"}\ \mathtt{"s"}.\ \mathtt{FUNPOW}\ \mathtt{(APP}\ \underline{\mathtt{"s"}}\mathtt{)}\ n\ \underline{\mathtt{"z"}})$$

The form $\mathtt{APP}\ M$ *is a partial application of the constructor for application terms, and* $\mathtt{FUNPOW}\ f\ n\ x$ *applies the function* f *to the* x *argument* n *times.*

Thus, $\mathtt{church}\ 3$ expands to

$$(\lambda\mathtt{"z"}\ \mathtt{"s"}.\ \underline{\mathtt{"s"}} \diamond (\underline{\mathtt{"s"}} \diamond (\underline{\mathtt{"s"}} \diamond \underline{\mathtt{"z"}})))$$

Definition 3. *Having used the same approach to model pairs (with constructor* `cpair` *and projections* `cfst` *and* `csnd`*), one can then define a recursion combinator:*

$$\begin{array}{l}
\vdash \mathtt{natrec}\ \mathtt{=} \\
\quad (\lambda\mathtt{"z"}\ \mathtt{"f"}\ \mathtt{"n"}. \\
\quad\quad \mathtt{csnd} \\
\quad\quad\ \ \diamond (\underline{\mathtt{"n"}} \diamond (\mathtt{cpair} \diamond \mathtt{church}\ 0 \diamond \underline{\mathtt{"z"}}) \\
\quad\quad\quad\quad \diamond (\lambda\mathtt{"r"}. \\
\quad\quad\quad\quad\quad\quad \mathtt{cpair} \diamond (\mathtt{csuc} \diamond (\mathtt{cfst} \diamond \underline{\mathtt{"r"}})) \\
\quad\quad\quad\quad\quad\quad\ \ \diamond (\underline{\mathtt{"f"}} \diamond (\mathtt{cfst} \diamond \underline{\mathtt{"r"}}) \diamond (\mathtt{csnd} \diamond \underline{\mathtt{"r"}})))))
\end{array}$$

The underlying recursion returns a pair of the original number and the actual desired result.

The characterising theorems are quite readable:

$$\begin{array}{l}
\vdash \mathtt{natrec} \diamond z \diamond f \diamond \mathtt{church}\ 0 \equiv_\beta z \\
\vdash \mathtt{natrec} \diamond z \diamond f \diamond \mathtt{church}\ (\mathtt{SUC}\ n) \equiv_\beta \\
\quad f \diamond \mathtt{church}\ n \diamond (\mathtt{natrec} \diamond z \diamond f \diamond \mathtt{church}\ n)
\end{array}$$

$$\begin{array}{l}
\vdash \texttt{cplus} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{church}\ (m + n) \\
\vdash \texttt{cminus} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{church}\ (m - n) \\
\vdash \texttt{cmult} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{church}\ (m \times n) \\
\vdash 0 < q \Rightarrow \texttt{cdiv} \diamond \texttt{church}\ p \diamond \texttt{church}\ q \to_n^* \texttt{church}\ (p\ \texttt{DIV}\ q)
\end{array}$$

$$\begin{array}{l}
\vdash \texttt{ceqnat} \diamond \texttt{church}\ n \diamond \texttt{church}\ m \to_n^* \texttt{cB}\ (n = m) \\
\vdash \texttt{cless} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{cB}\ (m < n)
\end{array}$$

$$\begin{array}{l}
\vdash \texttt{cfst} \diamond (\texttt{cpair} \diamond M \diamond N) \to_n^* M \\
\vdash \texttt{csnd} \diamond (\texttt{cpair} \diamond M \diamond N) \to_n^* N
\end{array}$$

Fig. 2. Theorems specifying the correctness of some of the various Church arithmetic and pair operations. The cB function takes a HOL boolean and returns the corresponding λ-term (either $(\boldsymbol{\lambda}\texttt{"x"}\ \texttt{"y"}.\ \underline{\texttt{"x"}})$ for true, or $(\boldsymbol{\lambda}\texttt{"x"}\ \texttt{"y"}.\ \underline{\texttt{"y"}})$ for false).

With a combinator of this sort, subtraction can be defined (and verified!) easily. (The traditional Church definition, which doesn't use pairing, is much harder to deal with.)

As well as the standard arithmetic operations (see Figure 2), we also need to define the minimisation operator, here called `cfindleast`. This is the only place where unbounded recursion, in the form of the Y combinator, is required. The introduction rule for a successful call is:

$$\begin{array}{l}
\vdash (\forall n.\ \exists b.\ P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB}\ b) \wedge P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB T} \Rightarrow \\
\quad \texttt{cfindleast} \diamond P \diamond k \equiv_\beta \\
\quad\quad k \diamond \texttt{church}\ (\texttt{LEAST}\ n.\ P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB T})
\end{array}$$

The preconditions require that

- the predicate P is total on numeric arguments, and also guaranteed to return a boolean on all such arguments; and
- the predicate P does indeed return true for at least one number.

The `LEAST` binder is the HOL analogue of `cfindleast`.

The k parameter to `cfindleast` is a continuation that is handed the result of a successful search for a number satisfying P. Using a continuation is a method for making functions that use minimisation *strict*. In other words, we want to be able to construct terms including minimisation, and to be sure that if the minimisation loops, then the whole term will have no β-normal form. The use of a continuation is the standard way to emulate call-by-value in a normal order setting. This insistence on strictness is consistent with the way we will handle the minimisation operator for recursive functions.

There is also an elimination rule for successful (terminating) `cfindleast` searches:

$$\begin{array}{l}
\vdash (\forall n.\ \exists b.\ P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB}\ b) \wedge \texttt{cfindleast} \diamond P \diamond k \equiv_\beta r \wedge \\
\quad \texttt{bnf}\ r \Rightarrow \\
\quad \exists m. \\
\quad\quad r \equiv_\beta k \diamond \texttt{church}\ m \wedge P \diamond \texttt{church}\ m \equiv_\beta \texttt{cB T} \wedge \\
\quad\quad \forall m_0.\ m_0 < m \Rightarrow P \diamond \texttt{church}\ m_0 \equiv_\beta \texttt{cB F}
\end{array}$$

The proof is by complete induction on the number of steps taken to reach the result r.

3 Reflection and the Universal Machine in the λ-Calculus

An important precursor to our computability results is the demonstration that the λ-calculus can implement itself.

3.1 Church de Bruijn Terms

The Church-style encoding of algebraic types is also possible for the algebraic type that encodes the "pure" de Bruijn terms (pdb): the type with three constructors dV, dAPP and dABS, of types num $\longrightarrow$ pdb, pdb $\longrightarrow$ pdb $\longrightarrow$ pdb and pdb $\longrightarrow$ pdb respectively. As noted above, we already know that the de Bruijn notion of β-reduction is isomorphic to that of the λ-calculus.

So we begin by defining an injection function from de Bruijn terms into λ-terms (cDB), along with "constructors" cdV, cdAPP and cdABS. We derive the following characterisations:

$$\vdash \texttt{cdV} \diamond \texttt{church}\ n \longrightarrow_n^* \texttt{cDB}\ (\texttt{dV}\ n)$$
$$\vdash \texttt{cdAPP} \diamond \texttt{cDB}\ M \diamond \texttt{cDB}\ N \longrightarrow_n^* \texttt{cDB}\ (\texttt{dAPP}\ M\ N)$$
$$\vdash \texttt{cdABS} \diamond \texttt{cDB}\ M \longrightarrow_n^* \texttt{cDB}\ (\texttt{dABS}\ M)$$

It is vital to be able to interpret de Bruijn terms at this point, rather than some sort of name-carrying syntax: with de Bruijn terms one does not have to implement variable-renaming when performing substitutions. Given the baggage of the Church encoding, the functions and terms developed here are already quite complicated enough without having to worry about some sort of gensym technology.

By analogy with natrec above, it is now possible to write a termrec recursion combinator for de Bruijn terms, with the following characterisation:

$$\vdash \texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ (\texttt{dV}\ i) \equiv_\beta v \diamond \texttt{church}\ i$$
$$\begin{aligned}\vdash\ & \texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ (\texttt{dAPP}\ t\ u) \equiv_\beta \\ & \quad c \diamond \texttt{cDB}\ t \diamond \texttt{cDB}\ u \diamond (\texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ t) \\ & \quad\ \diamond (\texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ u)\end{aligned}$$
$$\begin{aligned}\vdash\ & \texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ (\texttt{dABS}\ t) \equiv_\beta \\ & \quad a \diamond \texttt{cDB}\ t \diamond (\texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ t)\end{aligned}$$

With termrec defined, it is straightforward to define a function to implement normal-order reduction, and another to perform n steps of normal order reduction. With the minimisation operator, one can then define the function which finds the least n such that n steps of normal order reduction results in a term in β-normal form. Thus, we have a computable (and partial!) function for computing β-normal forms, which we call cbnf_ofk. As with cfindleast, the cbnf_ofk function takes a continuation parameter to help with strictness. We derive the following characterising theorems:

$$\begin{aligned}\vdash\ & \texttt{bnf\_of}\ M = \texttt{NONE} \Rightarrow \\ & \texttt{bnf\_of}\ (\texttt{cbnf\_ofk} \diamond k \diamond \texttt{cDB}\ (\texttt{fromTerm}\ M)) = \texttt{NONE}\end{aligned}$$
$$\begin{aligned}\vdash\ & \texttt{bnf\_of}\ M = \texttt{SOME}\ N \Rightarrow \\ & \texttt{cbnf\_ofk} \diamond k \diamond \texttt{cDB}\ (\texttt{fromTerm}\ M) \equiv_\beta k \diamond \texttt{cDB}\ (\texttt{fromTerm}\ N)\end{aligned}$$

$\vdash$ `cbnf_ofk` $\diamond$ k $\diamond$ `cDB` $M \rightarrow_n^* t' \wedge$ `bnf` $t' \Rightarrow$
$\exists M'$.
`bnf_of (toTerm` M`) = SOME (toTerm` M'`)` $\wedge$ k $\diamond$ `cDB` $M' \rightarrow_n^* t'$

The `bnf_of` function is the (uncomputable) function in the logic which, using an option type to encode partiality, returns a term's β-normal form if it has one. The `fromTerm` and `toTerm` functions are mutual inverses mapping from the λ-terms to the de Bruijn terms and *vice versa*.

3.2 The Universal Machine

In order to compare the λ-calculus's capabilities to what is done in other computational models, we restrict our attention to functions on natural numbers only. We also index the computable functions with natural numbers, so that we can define

$$\Phi : \texttt{num} \rightarrow \texttt{num} \rightarrow \texttt{num option}$$

taking parameters specifying the computable function to run, and the argument to run it on. The restriction to a single parameter for the given function is not significant because of the existence of standard encodings for lists and pairs of numbers.

The first parameter to Φ requires a bijection between the natural numbers and the de Bruijn terms. The HOL function `dBnum` is defined:

$\vdash$ `dBnum (dV` i`) = 3` $\times$ i
$\vdash$ `dBnum (dAPP` M N`) = 3` $\times$ `(dBnum` $M \otimes$ `dBnum` N`) + 1`
$\vdash$ `dBnum (dABS` M`) = 3` $\times$ `dBnum` M `+ 2`

(where $x \otimes y$ is a bijective pairing function on natural numbers). Its inverse, `numdB`, is defined by recursion on $\mathbb{N}$.

Definition 4. *The* Φ *function is defined:*

$\vdash$ Φ m n `=`
`OPTION_MAP force_num`
`(bnf_of (toTerm (numdB` m`)` $\diamond$ `church` n`))`

The `force_num` *function takes a* λ*-term and returns* n *if it is an instance of* `church` n*, and* `0` *otherwise.*

The Φ function gives us a purely HOL-level picture of the λ-calculus's computational capabilities, expressed in terms of functions on natural numbers. It will be our target when we investigate the capabilities of the recursive functions in Section 4 below.

Theorem 1. *There exists a* λ*-term that computes* Φ*. It is called* `UM`*, with characterising theorems:*

$\vdash$ Φ m n `= NONE` $\iff$ `bnf_of (UM` $\diamond$ `church (`$m \otimes n$`)) = NONE`
$\vdash$ Φ m n `= SOME` p $\iff$
`bnf_of (UM` $\diamond$ `church (`$m \otimes n$`)) = SOME (church` p`)`

$$\frac{}{\texttt{primrec zerof 1}} \qquad \frac{}{\texttt{primrec succ 1}}$$

$$\frac{i < n}{\texttt{primrec (proj } i\texttt{) } n}$$

$$\frac{\texttt{primrec } f\ |gs| \quad \texttt{EVERY (}\lambda g\texttt{. primrec } g\ m\texttt{) } gs}{\texttt{primrec (Cn } f\ gs\texttt{) } m}$$

$$\frac{\texttt{primrec } b\ n \quad \texttt{primrec } r\ \texttt{(}n\texttt{ + 2)}}{\texttt{primrec (Pr } b\ r\texttt{) (}n\texttt{ + 1)}}$$

Fig. 3. The primitive recursive functions. The relation `primrec` $f\ n$ is true if f is primitive recursive and behaves "sensibly" on arguments of length n (because HOL functions are total, f will have a value on lists of other lengths too). The auxiliaries are as follows: `zerof` is the constant function returning `0`; `succ` returns the successor of the head of a list; `proj` is the projection function on lists; `Cn` (composition) and `Pr` (primitive recursion) are described in the main text. The `EVERY` auxiliary is from HOL's theory of lists and checks a predicate holds of every element in a list.

4 The Recursive Functions

The first issue to resolve when modelling the recursive functions is whether to treat them "shallowly" or "deeply". This is not an issue that arises with the λ-calculus where it is natural to want to model the syntax of the calculus, and to then ascribe meaning to that syntax (a deep embedding). By way of contrast, with the recursive functions it seems equally natural to want to use the existing functions that exist in HOL, to identify a subset of those as primitive recursive, to then extend that subset with minimisation and thereby gain the recursive functions. Unfortunately, one then has to deal with the fact that the recursive functions are of variable arity, which is difficult to model in HOL's unsophisticated type system.

Rather than force the burden onto the type system, we use the type

$$\texttt{num list} \rightarrow \texttt{num}$$

for the primitive recursive functions, and add arity information to the inductive definition which identifies them. This approach doesn't treat application of a function to the wrong number of arguments (a list of the wrong length) as a type-error, but expects the sanity checking to be enforced through appropriate `primrec` assumptions (see Figure 3). The interesting auxiliary constants from that definition are for function composition (`Cn`) and primitive recursion (`Pr`), with characterising theorems:

$$\vdash \texttt{Cn } f\ gs\ \ell \texttt{ = } f\ \texttt{(MAP (}\lambda g\texttt{. } g\ \ell\texttt{) } gs\texttt{)}$$
$$\vdash \texttt{Pr } b\ r\ \texttt{(0::}t\texttt{) = } b\ t$$
$$\vdash \texttt{Pr } b\ r\ \texttt{(SUC } m\texttt{::}t\texttt{) = } r\ \texttt{(}m\texttt{::Pr } b\ r\ \texttt{(}m\texttt{::}t\texttt{)::}t\texttt{)}$$

Apart from all the standard arithmetic that can be shown to be primitive recursive, we gain confidence in this definition by also proving the famous result about Ackermann's function.

Theorem 2. *For any primitive recursive function f, there is an index J such that for all possible arguments xs, $f(xs)$ is always less than the Ackermann function applied to J and the sum of the values in xs:*

$$\begin{array}{l}\vdash \texttt{primrec}\ f\ k \Rightarrow \\ \quad \exists J.\ \forall xs.\ |xs| = k \Rightarrow f\ xs < \texttt{Ackermann}\ J\ (\texttt{SUM}\ xs)\end{array}$$

The proof closely follows the version of this result in the Isabelle/HOL sources, which is in turn based on Szasz [8].[2]

Recursive Functions. Adding minimisation to the primitive recursive functions forces the use of the option type to correctly model partiality. Thus the recursive functions are all of type

$$\texttt{num list} \to \texttt{num option}$$

Both the minimisation operation and the composition operator for recursive functions have rather ugly definitions (see Figure 4). The term $\texttt{minimise}\ f\ \ell$ is $\texttt{NONE}$ if there is no value x such that $f\ (x::\ell) = \texttt{SOME}\ 0$, or if there is some $y < x$ such that $f\ (y::\ell) = \texttt{NONE}$. Function composition is strict: if any of the functions in the list gs fails on the provided argument, so too does the composition. Similarly, primitive recursion: if a recursive call fails on $n < m$, then the recursive call on m must be held to fail as well.

The analogue of the `primrec` constant is `recfn`. It is an easy induction on the rules governing `primrec` to show

$$\vdash \texttt{primrec}\ f\ n \Rightarrow \texttt{recfn}\ (\texttt{SOME} \circ f)\ n$$

In the proofs to come, minimisation is only used once. All the other necessary operations were shown to be primitive recursive. This is implicitly a proof of Kleene's Normal Form theorem, stating that all recursive functions can be expressed as a composition of a primitive recursive function, minimisation and one other primitive recursive function.

5 Computational Equivalence

The "Easy" Direction Given the existence of `cfindleast`, and the general machinery of the Church numbers, one might imagine it straightforward to prove that the λ-calculus can implement the recursive functions. However, the journey is beset by a number of annoyances. First: how to represent the list of arguments the recursive functions expect? Our answer is to use the `nlist_of` function, which bijectively encodes a list of natural numbers as a single natural number.

Theorem 3. $\vdash \texttt{recfn}\ f\ n \Rightarrow \exists i.\ \forall \ell.\ \Phi\ i\ (\texttt{nlist\_of}\ \ell) = f\ \ell$

(The fact that the theorem quantifies over all *ℓ (rather than just those of length n) is a consequence of the fact that the definitions of the (primitive) recursive operators (*`Pr`, `Cn` *etc) actually give them reasonable values on lists of the wrong size. This can be emulated in the λ-calculus too.)*

[2] My original proof, in the Kananaskis-6 release, follows Taylor's more complicated argument [9]. Thanks to the anonymous referees for the pointer to the proofs in Isabelle/HOL and by Szasz.

```
⊢ recCn f gs ℓ =
    (let results = MAP (λg. g ℓ) gs
     in
       if EVERY (λr. r ≠ NONE) results then
         f (MAP THE results)
       else
         NONE)

⊢ minimise f ℓ =
    if
      ∃n.
        f (n::ℓ) = SOME 0 ∧
        ∀i. i < n ⇒ ∃m. 0 < m ∧ f (i::ℓ) = SOME m
    then
      SOME
        (εn.
           f (n::ℓ) = SOME 0 ∧
           ∀i. i < n ⇒ ∃m. 0 < m ∧ f (i::ℓ) = SOME m)
    else
      NONE
```

Fig. 4. The composition and minimisation operations for the recursive functions. As with primrec, these definitions are used in an inductive definition that specifies valid arities.

Proof. The big issue in this proof is the accurate modelling of partiality. For example, consider the primitive recursion case. By our inductive hypothesis, we have an i and j which are the indexes of the 0-case function and SUC-case function respectively. If the argument on which the function is recursing is n, it is necessary to set up a stack of n pending computations, linked together with continuation arguments. Thus, machine i is run first, and if it terminates, its result is passed to machine j with varying argument 0. This instance will have a continuation that passes the result onto machine j with varying argument 1, and so on, all the way up to one final computation: machine j with arguments $n - 1$, the result of the previous computation, and a continuation which is the identity function. With this nesting structure, the constructed λ-term is guaranteed to loop (fail) if and only if there is a failure in the calls made by the recursive function.

The Hard Direction. In the other direction, it is necessary to model the de Bruijn terms as numbers, and to perform all of the appropriate operations (*e.g.*, finding a redex, performing a substitution) purely arithmetically. Moreover, these operations are all shown to be primitive recursive, further increasing the complexity of the proofs and definitions.

As an example, the following theorems are the key facts about the form of substitution on de Bruijn terms (called nsub here) that simultaneously adjusts indices to reflect the disappearance of an outer abstraction (as happens in β-reduction). The first theorem states that the new constant pr_nsub really does the right thing with suitably encoded terms; the second that the constant really is primitive recursive:

```
⊢ pr_nsub [s; k; t] = dBnum (nsub (numdB s) k (numdB t))
⊢ primrec pr_nsub 3
```

The complexity in these proofs stem from the fact that we need to perform recursions that are not obviously primitive recursive. Firstly, when recursing over an encoded term, sub-terms have encodings that are numbers (much) smaller than the enclosing term, not just one less. Secondly, one also needs to be able to vary the accompanying parameters, as happens to k in the `dABS` clause of the definition of the `lift` function:

```
⊢ lift (dABS s) k = dABS (lift s (k + 1))
```

It is folklore that both of these variations do not require anything more than primitive recursion. Actually achieving them requires the use of primitive recursive functions that return large lists (encoded as numbers!) of results rather than single numbers. At the call-site, the calling function can then pick out the result it is really interested in, and then calculate an even larger list to be its own result.

When all this work within the primitive recursive functions has been done, the minimisation operation can be used to define the recursive "β normal form of" function, with definition:

```
⊢ recbnf_of =
    recCn (SOME ∘ pr_steps)
      [minimise (SOME ∘ pr_steps_pred); SOME ∘ proj 0]
```

The (primitive recursive) `pr_steps` function takes parameters n and t and performs n normal order reduction steps on t. The (primitive recursive) `pr_steps_pred` function takes parameters n and t and returns 0 if n steps of normal order reduction on t produces a term in β-normal form.

We are thus able to characterise `recbnf_of`:

```
⊢ recfn recbnf_of 1
⊢ recbnf_of [t] =
    OPTION_MAP (dBnum ∘ fromTerm) (bnf_of (toTerm (numdB t)))
```

This leads to

Theorem 4. *There exists a recursive function* `recPhi` *of type*

```
num list → num option
```

which emulates Φ*:*

```
⊢ recfn recPhi 2
⊢ recPhi [i; n] = Φ i n
```

6 Computability Theorems

Here we list a number of standard results that can be derived on top of the framework that has been established. The most complicated proofs are those to do with the recursively enumerable sets, where care is often required to handle computations that may not terminate.

Definition 5. *A* recursive *set (of natural numbers) is one that a computable function decides:*

$$\vdash \texttt{recursive}\ s \iff \exists m.\ \forall e.\ \Phi\ m\ e = \texttt{SOME}\ (\textbf{if}\ e \in s\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

Theorem 5. *The empty, finite and universal sets are recursive; recursive sets are closed under union, intersection and complement.*

$$\vdash \texttt{recursive}\ \emptyset$$
$$\vdash \texttt{recursive}\ \mathcal{U}(\texttt{:num})$$
$$\vdash \texttt{FINITE}\ s \Rightarrow \texttt{recursive}\ s$$
$$\vdash \texttt{recursive}\ s_1 \land \texttt{recursive}\ s_2 \Rightarrow \texttt{recursive}\ (s_1 \cup s_2)$$
$$\vdash \texttt{recursive}\ s_1 \land \texttt{recursive}\ s_2 \Rightarrow \texttt{recursive}\ (s_1 \cap s_2)$$
$$\vdash \texttt{recursive}\ (\texttt{COMPL}\ s) \iff \texttt{recursive}\ s$$

where $\mathcal{U}(\texttt{:num})$ *denotes the universal set of natural numbers, and where* COMPL *s is the complement of set s.*

Definition 6. *A* recursively enumerable *(r.e.) set is one that is the range of a computable function*

$$\vdash \texttt{re}\ s \iff \exists M_i.\ \forall e.\ e \in s \iff \exists j.\ \Phi\ M_i\ j = \texttt{SOME}\ e$$

Theorem 6. *Alternatively, the r.e. sets are those that are the domains of computable functions:*

$$\vdash \texttt{re}\ s \iff \exists N.\ \forall e.\ e \in s \iff \exists m.\ \Phi\ N\ e = \texttt{SOME}\ m$$

This result requires an implementation of dove-tailing, whereby the machine M_i is run on arguments $0..n-1$ for n steps, and the results examined for β-normal forms. If the argument e is not among them, then the process is repeated with parameter $n+1$.

Theorem 7. *All recursive sets are r.e. The r.e. sets are closed under union and intersection. If a set and its complement are r.e., then they are both recursive.*

$$\vdash \texttt{recursive}\ s \Rightarrow \texttt{re}\ s$$
$$\vdash \texttt{re}\ s \land \texttt{re}\ t \Rightarrow \texttt{re}\ (s \cap t)$$
$$\vdash \texttt{re}\ s \land \texttt{re}\ t \Rightarrow \texttt{re}\ (s \cup t)$$
$$\vdash \texttt{re}\ s \land \texttt{re}\ (\texttt{COMPL}\ s) \Rightarrow \texttt{recursive}\ s$$

Theorem 8. The Halting Problem. *Let* $\mathcal{K}$ *be defined as follows ("the machines that halt on their own indices"):*

$$\vdash \mathcal{K} = \{M_i \mid \exists z.\ \Phi\ M_i\ M_i = \texttt{SOME}\ z\}$$

Then $\mathcal{K}$ *is r.e. but not recursive. Its complement is not even r.e.*

$$\vdash \neg\texttt{recursive}\ \mathcal{K}$$
$$\vdash \texttt{re}\ \mathcal{K}$$
$$\vdash \neg\texttt{re}\ (\texttt{COMPL}\ \mathcal{K})$$

Theorem 9. The "s-1-1" theorem. *There exists a computable function with index* `s11` *that, when given an encoded pair $x \otimes y$, returns the index of a function that computes the function $\lambda z.\ \Phi\ x\ (y \otimes z)$. In other words, x is the index of the function to be partially evaluated with parameter y provided in advance.*

$$\vdash \forall x\ y.\ \exists f_i.\ \Phi\ \mathtt{s11}\ (x \otimes y) = \mathtt{SOME}\ f_i \wedge \forall z.\ \Phi\ f_i\ z = \Phi\ x\ (y \otimes z)$$

Theorem 10. The Recursion Theorem. *If f_i is the index of a total function (understood to be computing indices of other functions), then it has a fix-point e such that the functions with indices $f(e)$ and e are extensionally equal.*

$$\vdash (\forall n.\ \exists r.\ \Phi\ f_i\ n = \mathtt{SOME}\ r) \Rightarrow \exists e.\ \Phi\ (\mathtt{THE}\ (\Phi\ f_i\ e)) = \Phi\ e$$

(With the λ-calculus to hand, directly using the Y combinator is a much more pleasant prospect than the route *via* this theorem, with all its confusions of terms and indices encoding terms.)

Theorem 11. Rice's Theorem. *Let P be a predicate on functions. The predicate P is of type* `(num` $\rightarrow$ `num option)` $\rightarrow$ `bool` *and thus considers just the functions' extensional behaviour. Let* `indices` *P be the set of indices of computable functions satisfying P. Then, if* `indices` *P is recursive, that set is either the empty set, or the set of all numbers.*

$$\vdash \mathtt{recursive}\ (\mathtt{indices}\ P) \Rightarrow \mathtt{indices}\ P = \emptyset \vee \mathtt{indices}\ P = \mathcal{U}(\mathtt{:num})$$

7 Related Work

Zammit [11, §3] describes a HOL mechanisation of register machines, and shows that they can compute the recursive functions. He does not show the converse result. He also develops a Coq mechanisation of the recursive functions, and shows the *s-m-n* theorem in that model.

Computable functions of some form are necessarily a part of formalisations of Gödel's incompleteness theorems, and so mechanisations of that result by Shankar [7] and O'Connor [5] include approaches to computability. O'Connor uses the primitive recursive functions; Shankar uses a 'pure' subset of Lisp. Both are concerned with using their computational models to show that various formula manipulations are computable; neither is (directly) concerned with the limits of what is generally computable. Similarly, John Harrison's proof of Gödel's incompleteness theorem in the HOL Light system [2] focuses on showing the representability of primitive recursion in the embedded logic.

The Isabelle system comes with a mechanisation of the primitive recursive functions and a proof that Ackermann's function is not one of them. The ZF mechanisation is described in Paulson [6], who followed Szasz [8].

8 Conclusion

There is always more to do. Clearly, it would be appealing to mechanise the more operational models of computation: Turing and register machines. For the latter, the work by

Zammit may be a good starting point. If register machines are unappealing because of their general fiddliness, Turing machines are an even more daunting prospect. Nonetheless, the completist would clearly want to include both these models.

It would also be fun to attack further results in computability theory. For example, the theory of Turing degrees includes a number of classic results, with fascinating proofs. Alternatively, there is always basic complexity theory...

We cannot yet provide an easy route to proofs of computability for complicated systems with their own elaborate data types (as would be required for the introduction's motivating example). Nonetheless, the work done to date has demonstrated that the λ-calculus provides a good environment for working with rich types (such as the de Bruijn terms), and for manipulating them in ways known to be computable.

Acknowledgements. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Barendregt, H.P.: The Lambda Calculus: its Syntax and Semantics, revised edn. Studies in Logic and the Foundations of Mathematics, revised edition, vol. 103. Elsevier, Amsterdam (1984)
2. Harrison, J.: HOL Light system, `http://www.cl.cam.ac.uk/~jrh13/hol-light/`
3. Norrish, M.: Mechanising λ-calculus using a classical first order theory of terms with permutations. Higher Order and Symbolic Computation 19, 169–195 (2006)
4. Norrish, M., Vestergaard, R.: Proof pearl: De bruijn terms really do work. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 207–222. Springer, Heidelberg (2007)
5. O'Connor, R.: Essential incompleteness of arithmetic verified by coq. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 245–260. Springer, Heidelberg (2005)
6. Paulson, L.C.: A fixedpoint approach to implementing (co)inductive definitions. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 148–161. Springer, Heidelberg (1994)
7. Shankar, N.: Metamathematics, Machines and Gödel's Proof. Cambridge Tracts in Theoretical Computer Science, vol. 38. Cambridge University Press, Cambridge (1994)
8. Szasz, N.: A machine checked proof that Ackermann's function is not primitive recursive. In: Papers Presented at the Second Annual Workshop on Logical Environments, New York, NY, USA, pp. 317–338. Cambridge University Press, Cambridge (1993)
9. Gregory Taylor, R.: Ackermann's function is not primitive recursive, `http://home.manhattan.edu/~gregory.taylor/thcomp/pdf-files/ackerman.pdf` (Most recently accessed on 16, February 2011)
10. Urban, C., Cheney, J., Berghofer, S.: Mechanizing the metatheory of LF. ACM Transactions on Computational Logic 12, 1–42 (2011)
11. Zammit, V.: On the Readability of Machine Checkable Formal Proofs. PhD thesis, University of Kent at Canterbury (March 1999)

Automatic Differentiation in ACL2

Peter Reid[1] and Ruben Gamboa[2]

[1] University of Oklahoma, Norman, OK, USA
peter.d.reid@gmail.com,
http://www.cs.ou.edu

[2] University of Wyoming, Laramie, WY, USA
ruben@uwyo.edu,
http://www.cs.uwyo.edu/~ruben

Abstract. In this paper, we describe recent improvements to the theory of differentiation that is formalized in ACL2(r). First, we show how the normal rules for the differentiation of composite functions can be introduced in ACL2(r). More important, we show how the application of these rules can be largely automated, so that ACL2(r) can automatically define the derivative of a function that is built from functions whose derivatives are already known. Second, we show a formalization in ACL2(r) of the derivatives of familiar functions from calculus, such as the exponential, logarithmic, power, and trigonometric functions. These results serve as the starting point for the automatic differentiation tool described above. Third, we describe how users can add new functions and their derivatives, to improve the capabilities of the automatic differentiator. In particular, we show how to introduce the derivative of the hyperbolic trigonometric functions. Finally, we give some brief highlights concerning the implementation details of the automatic differentiator.

Keywords: ACL2, nonstandard analysis, automatic differentiation.

1 Introduction

ACL2(r) is a variant of the theorem prover ACL2 that offers support for reasoning about the irrational real and complex numbers via nonstandard analysis [8]. Since its logic is strictly first order and the theorem prover has only limited support for quantifiers, ACL2 would not appear to be a good candidate for reasoning about real analysis. However, by introducing key concepts from nonstandard analysis, such as “classical,” “standard part,” and the transfer principle, ACL2(r) extends ACL2 just enough to take advantage of its strong support for induction, which serves a key role in arguments using nonstandard analysis.

As a result, many theorems from real analysis have been formalized in ACL2, including the fundamental theorem of calculus [10] and several results having to do with differentiability [6,7]. However, much of this work is foundational in nature, while the intended use of ACL2(r) is to support reasoning about real-world software whose correctness relies on facts from basic engineering mathematics.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 312–324, 2011.

In this paper, we describe recent results that greatly expand the usefulness of ACL2(r) when reasoning about elementary functions and their derivatives. In Sect. 3, we present a new ACL2(r) “event” that lets the user introduce a theorem relating a function to its derivative. For example, the derivative of $\sqrt{1+x^2}$ can be introduced with the definition

```
(defderivative sqrt-1+x**2-derivative
  (acl2-sqrt (+ 1 (* x x))))
```

The event `defderivative` symbolically differentiates the given expression to obtain an expression for the derivative. It also proves the theorems that assert that the new expression is indeed the derivative of the old one. The implementation of `defderivative` relies on formalizing the familiar algebraic differentiation rules, such as $(f+g)'(x) = f'(x) + g'(x)$. This is similar to the approach used in [7], but with one key difference. The proof obligations required by the metatheorems in [7] are too unwieldy to be automated successfully. In Sect. 4, we present a different formalization that is much easier to automate when the derivative is known, as is the case when it is discovered using the algebraic differentiation rules. Of course, algebraic differentiation rules are of little use without a priori knowledge of *some* derivatives, i.e., a database of known derivatives. In Sect. 5, we show how the derivatives of many useful functions from calculus are formalized in ACL2(r). In particular, the exponential function had been defined in ACL2(r) since it was first developed, but its derivative was never determined. We report in this paper our recent formalization of this result in ACL2(r). Moreover, we use this result to find the derivatives of other functions, including the trigonometric functions. Finally, in Sect. 6, we show how a user can extend the database of known derivatives by proving a derivative fact, perhaps from first principles. In particular, we show how the user can introduce the hyperbolic trigonometric functions and their derivatives.

2 Related Work

Finding the derivative of function is a task that has many applications, such as optimization and sensitivity analysis. Consequently, many researchers have tackled the problem of automatically finding the derivative of a function expressed as a computer program. In fact, automatic differentiation (AD) is an established research area [1,4,9,5].

The approach used in AD is to compute the derivative of a program by examining the program statically. That is, the program's source code is transformed so that it can compute not only the original function, but also its derivative. This can be done either by using overloaded operators (in languages that support them), or by using preprocessing techniques to produce a new function. Naturally, this means that most solutions are program-specific, e.g., ADIC for C programs [3] and ADIFOR for FORTRAN programs [2], which use similar ideas but with different implementations.

Our interest is in finding the derivatives of functions expressed as programs in Common LISP. To that extent, our work is related to that in [11]. However, our primary interest is in automatically finding the *proof* that the derivative is correct, not just in finding the derivative. The techniques described in [11] go far beyond the work described in this paper as far as automatic differentiation, e.g., handing general derivatives of multivariate functions $f : R^n \rightarrow R^m$. But the emphasis there is in programming, not proving formal correctness using an automated theorem prover.

In spirit, our work has more in common with [12]. There, the concept of proof-carrying codes is applied to the AD transformations. The result is that the AD tool can produce a certificate that can be verified by a formal tool, thus establishing that the transformed function correctly computes the derivative of the input function. Our approach is quite different from that in [12] in that we are working with functional programs written in Common LISP, instead of abstract programs in a Hoare-style WHILE language.

3 Introducing the Defderivative Event

We begin our presentation by showing how `defderivative` looks to the end user. Consider the expression $\sqrt{1+x^2}$, and suppose that the user wants to introduce its derivative in ACL2(r). This is trivial to do with `defderivative`:

```
(defderivative sqrt-1+x**2-derivative
  (acl2-sqrt (+ 1 (* x x))))
```

`Defderivative` computes an expression for the derivative that is, of course, equivalent to $x/\sqrt{1+x^2}$. This expression is computed automatically using symbolic differentiation. `Defderivative` then introduces the theorem `sqrt-1+x**2-derivative` that shows that this expression is, in fact, the derivative of $\sqrt{1+x^2}$. This theorem is equivalent to the following ACL2(r) statement:

```
(defthm sqrt-1+x**2-derivative
  (implies (and (acl2-numberp x)
                (realp (+ 1 (* x x)))
                (< 0 (+ 1 (* x x)))
                (acl2-numberp y)
                (realp (+ 1 (* y y)))
                (< 0 (+ 1 (* y y)))
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (acl2-sqrt (+ 1 (* x x)))
                          (acl2-sqrt (+ 1 (* y y))))
                       (- x y))
                    (* (/ 1/2 (acl2-sqrt (+ 1 (* x x))))
                       (+ 0 (+ (* x 1) (* x 1)))))))
```

This theorem states that, for suitable and close-together x and y, the differential between x and y is close to the derivative at x. This notion of “closeness” comes from nonstandard analysis. A thorough description is provided in [8], but for the purposes of this work it suffices to understand that it means there is only an infinitesimal difference between the two terms.

The hypotheses in the theorem are formed by combining the hypotheses required by each of the various rules applied during symbolic differentiation. It is evident that this combination is "blind," as many of the hypotheses are trivially true. The expression that defines the derivative is also raw. I.e., it is formed by blindly following of the composition rules. We have experimented with using ACL2's rewriter to simplify the body of the derivative, but we have found that the simplified (according to ACL2) form rarely corresponds to the user's expectation. For example, ACL2(r) simplifies the term above to the following:

```
(+ (* 1/2 x (/ (acl2-sqrt (+ 1 (* x x)))))
   (* 1/2 x (/ (acl2-sqrt (+ 1 (* x x))))))
```

So we have found it better in practice to leave the formula discovered by automatic differentiation as is, and let the user provide a simpler definition, if she wishes. Typically, ACL2(r) can prove that these definitions are equivalent, so the function defined by the user is also shown to be the derivative of the original function. In this way, the user can choose the definition used, but avoid the tedious steps required to prove that it is the actual derivative. It is important to note that it is usually much easier to prove that these two functions are equal than to show that they are the derivative of the original function! For example, ACL2(r) can prove the following completely automatically:

```
(equal (* (/ 1/2 (acl2-sqrt (+ 1 (* x x))))
          (+ 0 (+ (* x 1) (* x 1))))
       (/ x (acl2-sqrt (+ 1 (* x x)))))
```

In turn, that makes it trivial to simplify the derivative of $\sqrt{1+x^2}$, so that it matches the user's expectations:

```
(defthm sqrt-1+x**2-derivative-clean
  (implies (and (realp x)
                (realp y)
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (acl2-sqrt (+ 1 (* x x)))
                          (acl2-sqrt (+ 1 (* y y))))
                       (- x y))
                    (/ x (acl2-sqrt (+ 1 (* x x))))))
  :hints (("Goal" :use (:instance
                         sqrt-1+x**2-derivative))))
```

Notice that we have simplified not only the formula for the derivative, but also the hypotheses.

This example demonstrates something fundamental about `defderivative`'s operation. We have not formally verified that `defderivative` produces correct derivatives in general, and ACL2's soundness does not rely on it doing so. Instead, `defderivative` makes a proof for ACL2 about a specific derivative when it is asked to. ACL2 verifies that this particular proof is correct, and the derivative is accepted. If `defderivative` had computed an incorrect derivative, ACL2 would have found the proof unconvincing and the whole operation would have had no effect, leaving the system's soundness intact. We chose this approach because it is more compatible with ACL2 as a first-order logic. Reasoning about `defderivative`'s operation as it is processing functions cannot be done simply in a first-order logic, since functions are not first-order objects.

4 The Implementation of Defderivative

4.1 Finding the Derivative

`Defderivative` can differentiate a function that is defined according to the following forms, where the derivative is taken with respect to the variable x:

- The identity function, i.e., x.
- A constant. This can be a literal number, a variable other than x, or a function of zero arguments.
- Addition, i.e., $f(x) + g(x)$.
- Multiplication, i.e., $f(x) \times g(x)$.
- Composition, i.e., $f(g(x))$.

In these forms, f and g are either functions whose derivatives have been previously determined and verified, or formulas that `defderivative` can differentiate recursively. Functional inverses, i.e. $f^{-1}(x)$ can be differentiated in a way compatible with `defderivative`, though through a different process; since a inverse function involves a single operation rather than an arbitrary combination of operations, it does not fit cleanly into the architecture of `defderivative`.

The list of forms does not include subtraction or division. This is because ACL2 defines these operations by using the corresponding inverses. So $f - g$ is really handled as $f + (-g)$, and we treat $(-g)$ as the composition of the functions unary minus and g. Once the derivatives of unary minus and unary division (i.e., the multiplicative inverse) are known, `defderivative` handles subtraction and division through the functional composition rule.

As `defderivative` computes the derivatives of functions defined using any of the given forms, it also proves the theorems that establish that the new expression is the derivative of the given function. We refer to these theorems as the *derivative theorems*. The most important of these theorems relates the function's differential between two `i-close` points to its derivative, which captures the nonstandard notion of derivative. Letting `F` be the function, `F-PRIME` its derivative as found using symbolic differentiation, and `DOMAIN-P` the domain over which `F` is defined and is differentiable, this theorem takes the following form:

```
(implies (and (DOMAIN-P x)
              (DOMAIN-P y)
              (standardp x)
              (i-close x y)
              (not (equal x y)))
         (i-close (/ (- (F x) (F y))
                     (- x y))
                  (F-PRIME x)))
```

The remaining derivative theorems play supporting roles, ensuring that the functon and its derivative are numeric, continuous, and finite.

Readers familiar with ACL2(r) may notice that the formal statement of differentiability given above differs from the one used in prior formalizations. There are two important differences:

- In earlier work, we separated the notions of derivative and differentiability. So the definition of differentiability was stated entirely in terms of F and not F-PRIME. The notions are equivalent, of course, but in the context of this work, the formal statement above is much more convenient, since we already have F-PRIME.
- The predicate DOMAIN-P describes the domain over which F is differentiable. In earlier work, we used intervals to define this domain. This has the advantage that we can quantify over intervals in a first-order logic, like ACL2's. However, treating this domain as a function adds flexibility and makes it easier to automate the process of defining the appropriate domain over which the algebraic differentiation rules are applicable, e.g. $f(x) \neq 0$.

For functions defined according to the forms described above, a calculus text would prescribe applying differentiation rules such as the sum rule, the product rule, and the chain rule. Each of these rules expresses the derivative of the whole (f and g combined) in terms of the derivative of its parts (f and g individually). I.e, these correspond to theorems involving general functions—higher order logic. Since it is a first-order logic, ACL2(r) does not deal in higher-order logic directly. However theorems such as these can be proved using ACL2(r)'s encapsulate feature. An encapsulate invocation lists function signatures followed by assumptions that describe how those functions behave. These assumptions are referred to as constraints. Proofs about the encapsulated functions can be constructed using the encapsulated assumptions. Finally, concrete functions can be substituted into those encapsulated function signatures in whatever proofs were constructed, as long as the encapsulated assumptions can be shown to hold for the concrete functions being substituted.

The algebraic differentiation rules are encapsulated as follows. The encapsulated functions are

- f, its derivative, and its domain;
- g, its derivative, and its domain; and
- the combined function (e.g., $f + g$), its derivative, and its domain.

The constraints in the encapsulate are as follows:

- f satisfies the derivative theorems.
- g satisfies the derivative theorems.
- The combined function is related to f and g in some way. For example, the sum rule is encapsulated using the constraint

```
(equal (f+g x)
       (+ (f x) (g x))
```

- The derivative of the combined function is related to f, g, and their derivatives in some way. For example, in the sum rule, the constraint has the form

```
(equal (f+g-prime x)
       (+ (f-prime x) (g-prime x)))
```

- The combined function is "type-safe." I.e., when the combined function (e.g., $f+g$) is evaluated on a number in its domain, its value depends on the value of the functions f and g applied to numbers on their respective domains.

Using these assumptions, the combination books proceed to prove the derivative theorems about the combined function. When the derivative of a specific combination needs to be proved, these theorems can be instantiated with the specific functions f and g.

This work is similar to the combination rules presented in [7]. In fact, originally we tried to use the theorems from [7], but we discovered that these were not amenable to automation. One problem was that the existing theorems allowed differentiation only over a single interval. That made it impossible to reason automatically about the derivative of tangent, for example. Another problem was in ease of application. The combination theorems in [7] never state the derivative except as the standard part of a small differential. This introduces complexity, since that small differential needs to be shown to behave as a derivative should. The new composition theorems take an expression for the derivative explicitly, which greatly simplifies their proofs. This comes at virtually no cost to `defderivative`, since it already computes expressions for the derivative.

4.2 Proof Structure

Composition rules are useful, but putting them together to verify the derivative of a complicated function can be prohibitively tedious. Each function application in the expression being differentiated requires several dozen lines of carefully written theorems to instantiate the appropriate compositions, adding up to hundreds of lines of proof for a typical expression. The root cause of this fact is that ACL2 has little support for higher-order functions and requires that virtually every step of a higher-order proof be explicitly pointed out to it. Fortunately, macros provide a way out. `Defderivative` composes the theorem code that the user otherwise would have had to and submits it to ACL2, making differentiation take a few lines rather than a few hundred. At the heart of the system is a function, named `differentiate-fn`, which we have added to the theorem prover. Its signature is (roughly) as follows.

Inputs:

1. Function expression. For example, this could be `(acl2-sqrt (+ 1 (* x x)))`.
2. Derivative name. This is the name of differentiated function and serves as a prefix for the derivative theorems.

Outputs:

1. The function's derivative.
2. The function's domain.
3. A proof script of the derivative theorems, showing that the derivative and domain expressions returned actually represent the function's derivative.

Recall from the beginning of Sect. 4.1 that there are several forms that a differentiable expression can take. `Differentiate-fn` has a branch for each of these cases. The first two cases, where the expression to differentiate is x or a fixed number, are relatively trivial to implement. `Differentiate-fn` simply returns a canned proof of the appropriate theorems, renamed according to the prefix requested, along with a canned derivative (1 or 0, respectively) and domain. The other cases, which involve f and g, are more interesting. The proofs they return take the following form.

1. Prove (recursively) the derivative theorems about f.
2. Prove (recursively) the derivative theorems about g.
3. Disable all theorems, except the derivative theorems of f and g. This allows us to limit ACL2's proof search, so that we can automate the rest of the proof.
4. Instantiate the appropriate combination theorems to prove the derivative theorems of the combined function.
5. These derivative theorems are the only ones introduced by `differentiate-fn`.

In short, these proofs recursively verify the derivatives of the two functions being composed and then combine those proofs by instantiating some of the combination proofs discussed in Sect. 4.1.

The derivative of a function is automatically recognized if the function has been registered with `defderivative`. In the following sections, we will describe how the original set of functions are registered, and how the user can register new functions.

We conclude this section with a simple example that shows `defderivative` in action. Imagine that `acl2-sqrt` has been registered and `defderivative` is then asked to differentiate `(acl2-sqrt (+ x 3))`. First, `defderivative` will use the proofs, provided on registration, concerning `acl2-sqrt`, its derivative, and its domain to fill in the first recursive section of the proof structure; this is simply a matter of renaming those proofs. Second, `defderivative` will recursively differentiate `(+ x 3)`. This will use the differentiation rules for sum, with $f(x) = x$ and $g(x) = 3$, and these functions will be differentiated recursively. Of course, their derivatives are trivial to compute, so `defderivative` combines them to find the derivative of `(+ x 3)`. Finally `defderivative` will use the theorems about the derivative of $f \circ g(x)$, using $f = \sqrt{x}$ and $g = x + 3$.

5 The Path to Elementary Functions

In this section, we will describe how we have seeded `defderivative` with the derivatives of several functions from elementary calculus. This list includes e^x, the natural logarithm, $\sqrt{x}$, sine, cosine, arcsine, arccosine, and arctangent. Other functions, such as tangent, can be derived automatically with `defderivative`, since they are defined using elementary operations over the built-in functions, e.g., $\tan(x) = \sin(x)/\cos(x)$. The proof effort required to establish the derivatives of these functions was significant. Fig. 1 shows how the proofs are based on one another.

In tackling these proofs, we used three different approaches. The first was to prove the derivative from first principles, i.e., algebraic manipulation of the differential into an expression that approaches the derivative as the difference becomes small. The second approach was to use earlier, simpler proofs to bootstrap later ones. For example, because ACL2(r) defines sine and cosine in terms of exponentials, one can use `defderivative` to differentiate sine's definition and then show that the derivative is cosine. Proofs with these approach tended to be trivial. The third approach was to take advantage of one function being the inverse of another. use `defderivative` to differentiate functions that are defined as the inverse of a differentiable function, e.g., $\ln(x)$.

The most difficult proof was the derivative of e^x. In some settings, this is a trivial result. For example, some calculus books show that the derivative of a^x is proportional to a^x, then define e as the unique real number such that the proportion is equal to 1. Others start by defining e^x using its Taylor expansion, then observe that this infinite series is its own derivative. But neither of these options were open to us. The function e^x is defined in ACL2(r) indirectly, using

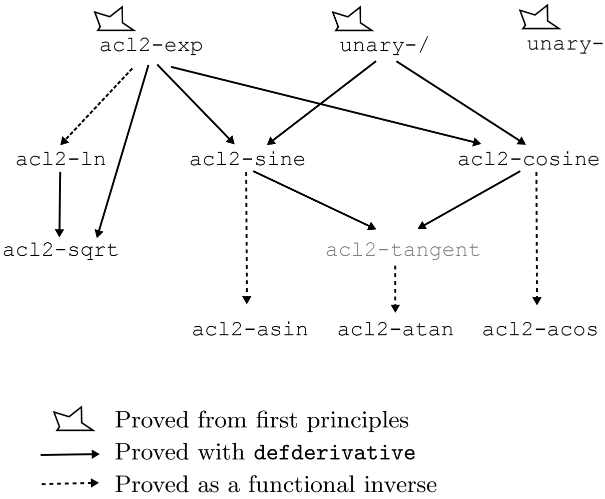


Fig. 1. Dependency graph of the functions built into `defderivative`. Symbols leading into a function represent how its derivative theorems were proved.

partial Taylor sums and the nonstandard transfer principle, so we could not rely on that proportion being 1 by definition. Moreover, the terms a^x that make up the Taylor expansion are defined in terms of e^x, so relying on a proof of the Taylor expansion's derivative would be circular. Instead, we had to follow a more direct approach.

To find the proof, we examined the value of the differential, $\frac{e^{x+\Delta x}-e^x}{\Delta x}$. Using the law of exponents, this reduces to $e^x\frac{e^{\Delta x}-1}{\Delta x}$. When x is standard, so is e^x, so it is sufficient to show that $\frac{e^{\Delta x}-1}{\Delta x} \approx 1$, i.e., is close to 1.

Proving that lemma was the biggest challenge. First, we defined $f(x) = \sum_{k=0}^{N} \frac{x^k}{(k+2)!}$. Then we showed that this series converged. That is, we showed that the partial sums are limited whenever x is limited, by comparing the partial sums with the Taylor expansion of e^x, which we had already shown converges. Since f converges, we can use the transfer principle to define the function $g(x) = {}^*f(x)$, the unique standard function that f converges to pointwise. It follows from the transfer principle and the definitions of g and e^x that $\frac{e^{\Delta x}-1}{\Delta x} = 1 + \Delta x \cdot g(\Delta x)$. Finally, we show that whenever $\Delta x \leq 1$, $||g(\Delta x)|| \leq ||g(1)||$ and therefore limited. So when Δx is infinitesimal, so is $\Delta x \cdot g(\Delta x)$, and it follows that $\frac{e^{\Delta x}-1}{\Delta x} = 1 + \Delta x \cdot g(\Delta x) \approx 1$.

6 Adding New Derivative Facts: Hyperbolic Trigonometric Functions

In this section, we show how a user can differentiate expressions involving a function that is not among those already registered with `defderivative`. To register a new function with `defderivative`, there are essentially two steps. First, the derivative theorems must be proved about that function. Second, `defderivative` must be informed of the new function, its derivative, its domain, and the associated proofs through a call to `def-elem-derivative`. This section provides an example of going through that process.

This example, in which we make `defderivative` able to differentiate hyperbolic sine and cosine, uses a bootstrapping approach. The hyperbolic functions are defined in terms of exponential functions, which `defderivative` already knows how to differentiate. The strategy will be to use `defderivative`'s existing capability to differentiate hyperbolic sine's definition and then to associate hyperbolic sine itself with the resulting derivative.

Hyperbolic sine and cosine and their derivatives are defined as follows:

$$\begin{aligned} sinh(x) &= \frac{e^x - e^{-x}}{2} & \qquad \frac{d}{dx}sinh(x) &= cosh(x) \\ cosh(x) &= \frac{e^x + e^{-x}}{2} & \qquad \frac{d}{dx}cosh(x) &= sinh(x) \end{aligned}$$

These definitions are trivial to enter in ACL2(r).

```
(defun acl2-sinh (x)
  (/ (- (acl2-exp x) (acl2-exp (- x)))
     2))

(defun acl2-cosh (x)
  (/ (+ (acl2-exp x) (acl2-exp (- x)))
     2))
```

Next, we use `defderivative` to find the derivative of the body of `acl2-sinh`.

```
(defderivative acl2-sinh-lemma
  (/ (- (acl2-exp x) (acl2-exp (- x)))
     2))
```

As expected, this results in an unsimplified domain and derivative. However, we can simplify it and introduce the derivative of `acl2-sinh` with the following theorem:

```
(defthm acl2-sinh-derivative
  (implies (and (acl2-numberp x)
                (acl2-numberp y)
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (acl2-sinh x)
                          (acl2-sinh y))
                       (- X Y))
                    (acl2-cosh x)))
  :hints (("Goal" :use (:instance acl2-sinh-lemma))))
```

That is the most difficult of the proof obligations that the user must prove before she can register the derivative of hyperbolic sine. The other obligations concern the remaining derivative theorems, and those are far simpler, such as showing that values in the domain of hyperbolic sine does not require call outsides the domain of e^x. Once these obligations are established, the user can register the derivative with the following event:

```
(def-elem-derivative
  acl2-sinh              # function to differentiate
  elem-acl2-sinh         # prefix of theorems' name
  (acl2-numberp x)       # domain
  (acl2-cosh x))         # derivative
```

7 Conclusions

In this paper, we described the macro `defderivative` and its implementation. This macro symbolically differentiates ACL2(r) expressions involving functions whose derivatives have been established previously, e.g., built-in functions,

functions derived using `defderivative`, and functions registered by the user. The macro also computes the appropriate domain for the function and proves the required derivative theorems.

In the process of implementing `defderivative`, we identified some impediments to automation in our previous treatment of algebraic differentiation rules, and we addressed those shortcomings as part of this project. The resulting framework is much easier to use, hence more widely applicable. For example, the previous work was foundational, allowing one to prove (often tediously) when one function was the derivative of another. There were very few practical results. While the derivative of x^n was formalized in ACL2(r), that of e^x was not, nor were those of the trigonometric functions. We proved the derivative of e^x using techniques similar to the ones used previously with ACL2(r), but the remaining derivatives were derived automatically.

The macro `defderivative` can be readily extended to compute partial derivatives. However, the treatment of differentiation in ACL2(r) is derived from nonstandard analysis, and this imposes technical restrictions on the treatment of free variables. The result is that we must anticipate the number of variables that will be required. Thus far, we have implemented partial derivatives for functions of two variables, such as `expt`, which can represent either a^x or x^n, depending on which variable is fixed. To generalize this to functions of three or more variables, it will be more convenient to use a classical notion of derivative, i.e., one based on limits instead of standard part. We are currently working on a proof of the equivalence of these notions in ACL2(r). However, the proof is quite challenging, because it involves quantifiers and infinite sets, neither of which is supported well by ACL2(r).

References

1. Community portal for automatic differentiation, `http://www.autodiff.org`
2. Corliss, G., Griewank, A., Bischof, C., Carle, A., Hovland, P.: ADIFOR: Generating derivative codes from fortran programs. Scientific Programming (1) (1991)
3. Roh, L., Bischof, C., Mauer-oats, A.: ADIC: An extensible automatic differentiation tool for ANSI-C 27, 1427–1456 (1997)
4. Hovland, P.D., Bischof, C.H., Norris, B.: On the implementation of automatic differentiation tools. Higher Order Symbol. Comput. 21, 311–331 (2008)
5. Corliss, G., Faure, C., Griewank, A., Hascoet, L., Naumann, U. (eds.): Automatic Differentiation of Algorithms: From Simulation to Optimization. CIS. Springer, Heidelberg (2001); Selected papers from the AD2000 conference, Nice, France (June 2000)
6. Gamboa, R.: Continuity and differentiability in ACL2. In: Kaufmann, M., Manolios, P., Moore, J.S. (eds.) Computer-Aided Reasoning: ACL2 Case Studies, ch. 18. Kluwer Academic Press, Dordrecht (2000)
7. Gamboa, R., Cowles, J.: The chain rule and friends in ACL2(r). In: Proceedings of the Eighth International Workshop of the ACL2 Theorem Prover and its Applications, ACL2 2009 (2009)
8. Gamboa, R., Kaufmann, M.: Nonstandard analysis in ACL2. Journal of Automated Reasoning 27(4), 323–351 (2001)

9. Griewank, A., Walthe, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Other Titles in Applied Mathematics. SIAM, Philadelphia (2008)
10. Kaufmann, M.: Modular proof: The fundamental theorem of calculus. In: Kaufmann, M., Manolios, P., Moore, J.S. (eds.) Computer-Aided Reasoning: ACL2 Case Studies, ch. 6, Kluwer Academic Press, Dordrecht (2000)
11. Pearlmutter, A.B., Siskind, J.M.: Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM Trans. Program. Lang. Syst. 30, 7:1–7:36 (2008)
12. Tadjouddine, E.M.: On formal certification of AD transformations. In: Barth, T.J., et al. (eds.) MFCS 1978. LNCSE, vol. 64, pp. 23–33. Springer, Heidelberg (2008)

seL4 Enforces Integrity*

Thomas Sewell[1], Simon Winwood[1,2], Peter Gammie[1], Toby Murray[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
{first-name.last-name}@nicta.com.au

Abstract. We prove that the seL4 microkernel enforces two high-level access control properties: integrity and authority confinement. Integrity provides an upper bound on write operations. Authority confinement provides an upper bound on how authority may change. Apart from being a desirable security property in its own right, integrity can be used as a general framing property for the verification of user-level system composition. The proof is machine checked in Isabelle/HOL and the results hold via refinement for the C implementation of the kernel.

1 Introduction

Enforcing access control is one of the primary security functions of an operating system (OS) kernel. Access control is usually defined as two properties: confidentiality, which means that information is not acquired without read authority, and integrity, which means that information is not modified without write authority. These properties have been well studied in relation to classical security designs such as the Bell-LaPadula model [3]. For dynamic access control systems, such as the capability system in the seL4 microkernel, an additional property is of interest: authority confinement, which means that authority may not be spread from one subject to another without explicit transfer authority.

We have previously verified the functional correctness of seL4 [12]. In this work we prove that seL4 correctly enforces two high level security properties: integrity and authority confinement.

We define these properties with reference to a user-supplied security policy. This policy specifies the maximum authority a system component may have. Integrity limits state mutations to those which the policy permits the subject components to perform. Authority confinement limits authority changes to those where components gain no more authority than the policy permits. The policy provides mandatory access control bounds; within these bounds access control is discretionary.

While integrity is an important security property on its own, it is of special interest in formal system verification. It provides a framing condition for the

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 325–340, 2011.

execution of user-level components, telling us which parts of the system do not change. In a rely-guarantee framework, the integrity property gives us useful guarantee conditions for components without the need to consult their code. This is because the kernel, not the component, is providing the guarantee. This becomes especially important if the system contains components that are otherwise beyond our means for formal code-level verification, such as a Linux guest operating system with millions of lines of code. We can now safely and formally compose such parts with the rest of the system.

Access control properties and framing conditions have been extensively studied. Proving these properties about a real OS kernel implementation, however, has not been achieved before [10]. Specifically, the novelty and contributions of this work are:

- The Isabelle/HOL formalisation and generalisation of integrity and authority confinement for a real microkernel implementation.
- To the best of our knowledge the first code-level proof of high-level access control properties of a high-performance OS kernel.

Our proof is connected to reality via refinement to the C implementation. This means we must deal with all the complexities and corner cases of the kernel we have, rather than laying out a kernel design which fits neatly with our desired access control model and hoping to implement it later.

We make one kind of simplifying assumption: we place restrictions on the policy. These forbid some kinds of interaction between components which are difficult for us to reason about. Although we have not yet applied the theorem to a large system, our choice of assumptions has been guided by a previous case study [2] on a secure network access device (SAC), with a dynamic and realistic security architecture.

We are confident that a significant variety of security designs will, after some cosmetic adjustments, comply with our restrictions. We support fine grained components, communication between them via memory sharing and message passing, delegation of authority to subsystems and dynamic creation and deletion of objects. We support but restrict propagation of authority and policy reconfiguration at runtime.

In the following, Sect. 2 gives a brief introduction to access control in general and to the seL4 access control system in particular. Sect. 2 also introduces a part of the aforementioned SAC system as a running example. Sect. 3 gives a summary of the formalisation as well as the final Isabelle/HOL theorems and Sect. 4 discusses the results together with our experience in proving them.

2 Access Control Enforcement and seL4

This section introduces relevant concepts from the theory of access control and our approach to instantiating them for seL4. For ease of explanation we will first introduce a running example, then use it to describe the seL4 security mechanisms available, and then compare to the theory of access control.

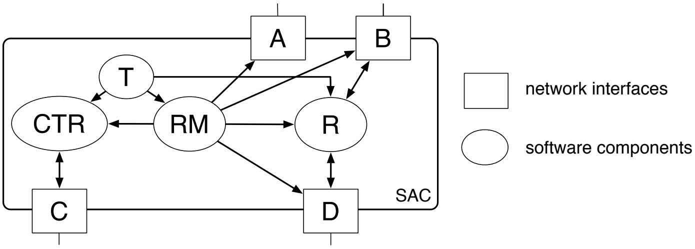


Fig. 1. System snapshot, routing between data D and back-end B network

2.1 Example

The secure access controller (SAC) was designed in an earlier case study of ours [2] and will serve as a running example in this work. The purpose of the SAC is to switch one front-end terminal between different back-end networks of separate classification levels. The security goal of the system is to avoid information flow between the back-end networks.

Fig. 1 shows the main components of the SAC as nodes and their logical authority connections as edges. To the outside, the SAC provides four network interfaces: two back-end networks A and B, one control network C, and one data network D. Networks C and D are attached to a front-end terminal. The purpose of the SAC is to connect either A to D or B to D at a time without information flow between A and B. Internally, we have four components: a timer T, a controller user interface CTR, the router manager RM, and a router instance R. The router manager RM will upon a switch request, tear down R, remove all access from it, create and start a fresh R component, and connect it to the requested networks. RM is a small, trusted component and has access to all networks. Verification needs to show that it does not abuse this access. The router R, on the other hand, is a large, untrusted instance of Linux. It will only ever be given access to one of the back-end networks during its lifetime.

In previous work [2], we have shown that, assuming a specific policy setup, assuming correct behaviour of RM, and assuming that the kernel correctly enforces access control, the security goal of the system will be enforced.

The work in this paper helps us to discharge the latter two assumptions: we can use the integrity property as a framing condition for verifying the behaviour of RM, and we can use the same property to make sure that R stays within its information flow bounds. To complete the verification, we would additionally need the confidentially side of access control as well as a certified initial policy set up. Both are left for future work.

2.2 seL4

The seL4 microkernel is a small operating system kernel. As a microkernel, it provides a minimal number of services to applications: interprocess communication, threads, virtual memory, access control, and interrupt control.

As mentioned, seL4 implements a capability-based access control system [6]. Services, provided by a set of methods on kernel implemented objects, are invoked by presenting to the kernel a capability that refers to the object in question and has sufficient access rights for the requested method. For the purposes of this paper, the following four object classes are the most relevant.

CNodes. Capabilities are stored in kernel-protected objects called *CNodes.* These CNodes can be composed into a *CSpace*, a set of linked CNodes, that defines the set of capabilities possessed by a single thread. CNode methods allow copying, insertion and removal of capabilities. For a thread to use a capability, this capability must be stored in the thread's CSpace. CNodes can be shared across CSpaces. The links in Fig. 1 mean that the collective CSpaces of a component provide enough capabilities to access or communicate with another component.

Virtual Address Space Management. A virtual address space in seL4 is called a *VSpace.* In a similar way to CSpaces, a VSpace is composed of objects provided by the microkernel. On ARM and Intel IA32 architectures, the root of a VSpace consists of a *Page Directory* object, which contains references to *Page Table* objects, which themselves contain references to *Frame* objects representing regions of physical memory. A Frame can appear in multiple VSpaces, and thereby implement shared memory between threads or devices such as the SAC networks.

Threads. Threads are the unit of execution in seL4, the *subjects* in access control terminology. Each thread has a corresponding *TCB* (thread control block), a kernel object that holds its data and provides the access point for controlling it. A TCB contains capabilities for the thread's CSpace and VSpace roots. Multiple threads can share the same CSpace and VSpace or parts thereof. A component in the SAC example may consist of one or more threads.

Inter-process Communication (IPC). Message passing between threads is facilitated by *Endpoints* (EP). The kernel provides *Synchronous Endpoints* with rendezvous-style communication, and *Asynchronous Endpoints* (AEP) for notification messages. Synchronous endpoints can also be used to transfer capabilities if the sender's capability to the endpoint has the *Grant* right. The edges in the SAC example of Fig. 1 that are physical communication links, are implemented using endpoints.

The mechanisms summarised above are flexible, but low-level, as customary in microkernels. Fig. 2 shows parts of the implementation of the link between the CTR and RM components of Fig. 1. The link is implemented via a synchronous endpoint EP. The CTR and RM components both consist of one main thread, each with a CSpace containing a capability to the endpoint (among others), and each with a VSpace containing page directories (pd), page tables (pt), and frames f_n implementing private memory.

These mechanisms present a difficulty for access control due to their fine grained nature. Once larger components are running, there can easily exist hundreds of thousands of capabilities in the system. In addition, seL4 for ARM has

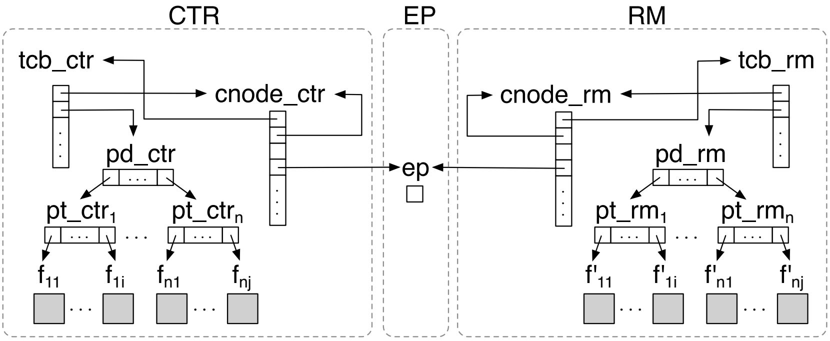


Fig. 2. SAC capabilities (partial)

15 different capability types, several of which have specific rights and variations. We will prune this complexity down by making some simplifying observations. One such observation is that many kinds of capabilities will not be shared without the sharing components trusting each other. For example sharing part of a CSpace with an untrusted partner makes little sense, as seL4 does not provide a mechanism for safely using a capability of unknown type.

2.3 Access Control Enforcement

An access control system controls the access of *subjects* to *objects* [14], by restricting the operations that subjects may perform on objects in each state of the system. As mentioned above, in seL4 the subjects are threads, the objects are all kernel objects, including memory pages and threads themselves.

The part of the system state used to make access control decisions, i.e., the part that is examined by the kernel to decide which methods each subject may perform on which object, is called the *protection state*. In seL4, this protection state is mostly represented explicitly in the capabilities present in the CSpace of each subject. This explicit, fine-grained representation is one of the features of capability-based access control mechanisms. In reality, however, some implicit protection state remains, for instance encoded in the control state of a thread, or in the presence of virtual memory mappings in a VSpace.

The protection state governs not only what operations are allowed to be performed, but also how each subject may modify the protection state. For instance, the authority for capabilities to be transmitted and shared between subjects is itself provided by CNode or endpoint capabilities.

As seen previously in Fig. 2, the protection state of a real microkernel can be very detailed, and therefore cumbersome to describe formally. It is even more cumbersome to describe precisely what the allowed effects of each operation are at this level. Hence we make use of the traditional concept of a policy which can be seen as an abstraction of the protection state: we assign a label to each object and subject, and we specify the authority between labels as a directed graph. The abstraction also maps the many kinds of access rights in seL4 protection states

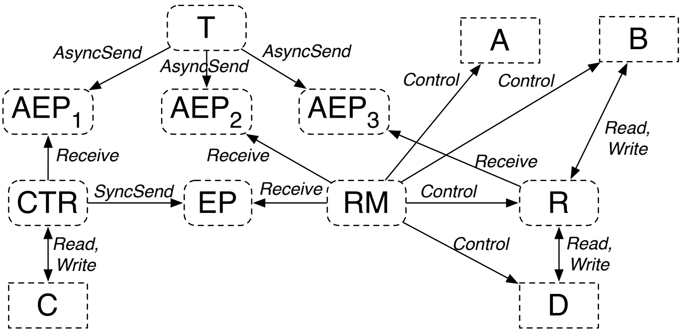


Fig. 3. SAC authority (except self-authority)

into a simple enumerated set of authority types. This simplifies the setup in three ways: the number of labels can be much smaller than the number of objects in the system, the policy is static over each system call whereas the protection state may change and, finally, we can formulate which state mutations are allowed by consulting the policy, rather than the more complex protection state.

The abstraction level of the policy is constrained only by the wellformedness assumptions we make in Sect. 3. Within these bounds it can be chosen freely and suitably for any given system.

Fig. 3 shows an abstract policy, only mildly simplified for presentation, that corresponds to a possible protection state of the SAC at runtime. The objects in the system are grouped by labels according to the intention of the component architecture. The RM label, for instance, includes all objects in the RM box of Fig. 2. The communication endpoints between components have their own label to make the direction of information flow explicit. The edges in the figure are annotated with authority types described in Sect. 3.

Correct access control enforcement, with respect to a security policy, can be decomposed into three properties about the kernel: *integrity*, *confidentiality* and *authority confinement*. The two former properties are the same as introduced in Sect. 1, only the notion of what is permitted is taken from the policy rather than the protection state. The latter property ensures the current subject cannot escalate its authority (or another subject's) above what the policy allows.

Note that we may have some components in the system, such as RM in the SAC, which have sufficient authority to break authority confinement. The authority confinement theorem will assume these components are not the current subject, and we will be obliged to provide some other validation of their actions.

3 Formalisation of Integrity Preservation

This section sketches our Isabelle/HOL formalisation of the integrity and authority confinement properties. While it is impossible in the space constraints of a paper to give the full detail, we show the major definitions and the top level theorems to provide a flavour of the formalisation. For formal details on the kernel-level concepts beyond our description below we refer the interested reader to the published Isabelle/HOL specification of the kernel [1] that the definitions here build on.

We have already introduced the notion of a policy, an upper bound on the protection state of the system, and an accompanying abstraction, a mapping from detailed subject and object names up to a smaller set of component labels. We roll these objects together with a subject label into the PAS record (policy, abstraction, subject) which is an input to all of our access control predicates.

$$\begin{array}{lrl} \textbf{record}\ 'l\ \mathsf{PAS}\ = & \mathsf{pasPolicy} :: & ('l \times \mathsf{auth} \times 'l)\ \mathsf{set} \\ & \mathsf{pasObjectAbs} :: & \mathsf{obj\text{-}ref} \Rightarrow 'l \\ & \mathsf{pasIRQAbs} :: & \mathsf{irq} \Rightarrow 'l \\ & \mathsf{pasASIDAbs} :: & \mathsf{asid} \Rightarrow 'l \\ & \mathsf{pasSubject} :: & 'l \end{array}$$

The type parameter $'l$ here is any convenient type for component labels. The policy field is a graph (a set of triples $'l \times \mathsf{auth} \times 'l$), whose vertices are policy labels and whose edges are authority types from the type auth which will be discussed shortly. Abstraction functions are provided for seL4's namespaces: objects (i.e. system memory), interrupt request numbers and address space identifiers. Each of these is mapped up to a policy label.

The subject in the PAS record identifies the label of the current subject. We must associate all (write) actions with a subject in order to define the integrity property we are proving. We will pick the subject associated with any kernel actions at kernel entry time, choosing the label of the currently running thread. This coarse division of actions between subjects causes some problems for message transfers, as will be discussed below.

The identification of a subject makes all of our access control work *subjective*. The integrity property depends on which subject carries out an action, because whether that action is allowed or not depends on the allowed authority of the subject performing it. Wellformedness of PAS records, encapsulating our policy assumptions, will be defined in a subjective manner as well, so for any given system and policy the authority confinement proof may be valid only for less-trusted subjects which satisfy our policy assumptions.

3.1 Authority Types

We made the observation in Sect. 2 that most kinds of objects in seL4 are not shared by mutually distrusting components. We found that the objects that could be safely shared were those where authority to that object could be partitioned between capabilities. Endpoints are a good example: Capabilities to endpoints can be send-only or receive-only. Every endpoint in the SAC has a single sending component and a single receiving component, as is typical in seL4 system architectures. Memory frames may also be read only or read-write and a typical sharing arrangement has a single writer, though possibly many readers.

This led us to our chosen formalisation of authority types.

$$\begin{aligned}\textbf{datatype}\ \mathsf{auth} = {} & \mathsf{Receive} \mid \mathsf{SyncSend} \mid \mathsf{AsyncSend} \mid \mathsf{Reset} \mid \mathsf{Grant} \\ & \mid \mathsf{Write} \mid \mathsf{Read} \mid \mathsf{Control}\end{aligned}$$

The Receive, Read and Write authorities have been described above. We distinguish endpoint types via SyncSend and AsyncSend. This distinction anticipates future work on confidentiality, where synchronous sends spread information in both directions. Capabilities to endpoints may also have the Grant right which permits other capabilities to be sent along with messages. The Reset authority is conferred by all capabilities to endpoints, since these capabilities can sometimes cause an endpoint reset even if they have zero rights. Finally, the Control authority is used to represent complete control over the target; it is a conservative approximation used for every other kind of authority in the system.

3.2 Subjective Policy Wellformedness

We aim to show authority is confined by a policy. This will not be true for all kinds of policies. If a policy gives the subject a Grant authority to any other component in addition to some authority which the other component should not have, that policy can clearly be invalidated. We define wellformedness criteria on policies, and thereby system architectures, given a specific subject, as follows. In our example Fig. 3, we would expect the policy to be wellformed for the subjects CTR, T, and R, but not RM.

$$\begin{aligned}
&\mathsf{policy\text{-}wellformed}\ \mathit{policy}\ \mathit{irqs}\ \mathit{subject} \equiv \\
&\quad (\forall\, a.\ (\mathit{subject},\ \mathsf{Control},\ a) \in \mathit{policy} \longrightarrow \mathit{subject} = a) \wedge \\
&\quad (\forall\, a.\ (\mathit{subject},\ a,\ \mathit{subject}) \in \mathit{policy}) \wedge \\
&\quad (\forall\, s\ r\ \mathit{ep}. \\
&\qquad (s,\ \mathsf{Grant},\ \mathit{ep}) \in \mathit{policy} \wedge (r,\ \mathsf{Receive},\ \mathit{ep}) \in \mathit{policy} \longrightarrow \\
&\qquad (s,\ \mathsf{Control},\ r) \in \mathit{policy} \wedge (r,\ \mathsf{Control},\ s) \in \mathit{policy}) \wedge \\
&\quad (\forall\, i{\in}\mathit{irqs}.\ \forall\, p.\ (i,\ \mathsf{AsyncSend},\ p) \in \mathit{policy} \longrightarrow (\mathit{subject},\ \mathsf{AsyncSend},\ p) \in \mathit{policy})
\end{aligned}$$

The first requirement is that the subject cannot have Control authority over another component. If it did there would be no point in separating these components, as the subject might coerce the target into taking actions on its behalf.

The second requirement is that the subject has all kinds of authority to itself. We always consider components permitted to reconfigure themselves arbitrarily.

The Grant restriction observes that successful capability transfers over messages are as problematic for access control as Control authority. In each direction this restriction could be lifted if we introduced more complexity.

In the sending direction the problem is that the sender can transfer an arbitrary capability into the receiver's capability space, giving the sender the new authority to rescind capabilities from the receiver's capability space in the future. It may be possible in seL4 for a receiver to partition its capability space to make this safe, but we know of no use case that justifies the resulting complexity.

In the receiving direction the problem is in the way we fix the subject of the message send. Synchronous sends in seL4 complete when both sender and receiver are ready. If the sender is ready when our subject makes a receive system call, it may appear that the receiver has broken authority confinement by magically acquiring new authority. In fact the authority belonged to the sender, which was involved as a subject in some sense, but not in a manner that is easy to capture.

The final policy assumption relates to interrupts. Interrupts may arrive at any time, delivering an asynchronous message to a thread waiting for that interrupt. We must allow the current subject to send this message. We hope to revisit our simple notion of the current subject in future work.

3.3 Policy/Abstraction Refinement

We define a kernel state s as being a refinement of the policy p as follows.

$$\begin{array}{l}
\mathsf{pas\text{-}refined}\ p\ s \equiv \\
\quad \mathsf{policy\text{-}wellformed}\ (\mathsf{pasPolicy}\ p)\ (\mathsf{range}\ (\mathsf{pasIRQAbs}\ p))\ (\mathsf{pasSubject}\ p) \wedge \\
\quad \mathsf{irq\text{-}map\text{-}wellformed}\ p\ s \wedge \\
\quad \mathsf{auth\text{-}graph\text{-}map}\ (\mathsf{pasObjectAbs}\ p)\ (\mathsf{state\text{-}objs\text{-}to\text{-}policy}\ s) \subseteq \mathsf{pasPolicy}\ p \wedge \\
\quad \mathsf{state\text{-}asids\text{-}to\text{-}policy}\ p\ s \subseteq \mathsf{pasPolicy}\ p \wedge \\
\quad \mathsf{state\text{-}irqs\text{-}to\text{-}policy}\ p\ s \subseteq \mathsf{pasPolicy}\ p
\end{array}$$

The kernel state refines the policy if the various forms of authority contained within it, when labelled by the abstraction functions, are a subset of the policy. The full definitions of the extraction functions for authority from the kernel state are too detailed to describe here. In summary, a subject has authority over an object for one of these reasons:

- it possesses a capability to the object.
- it is a thread which is waiting to conclude a message send. For performance reasons the capability needed to start the send is not rechecked on completion, and thus the thread state is an authority in its own right.
- it possesses the parent capability in the capability derivation tree (cdt) of a capability stored in the object.
- the page tables link the subject to the object.
- the active virtual address space database names the object as the page directory for an address space identifier the subject owns.
- the interrupt despatch mechanism lists the object as the receiver for an interrupt request number the subject owns.

Note that none of this authority extraction is subjective. The pas-refined predicate is subjective only because it also asserts policy-wellformed. The reason for this is convenience: these properties are almost always needed together in the proof.

3.4 Specification of Integrity

We define access control integrity subjectively as follows:

$$\begin{array}{l}\mathsf{integrity}\ p\ s\ s' \equiv \\ \quad (\forall x.\ \mathsf{object\text{-}integrity}\ p\ (\mathsf{pasObjectAbs}\ p\ x)\ (\mathsf{kheap}\ s\ x)\ (\mathsf{kheap}\ s'\ x)) \wedge \\ \quad (\forall x.\ \mathsf{memory\text{-}integrity}\ p\ x\ (\mathsf{tcb\text{-}states\text{-}of\text{-}state}\ s)\ (\mathsf{tcb\text{-}states\text{-}of\text{-}state}\ s') \\ \qquad (\mathsf{auth\text{-}ipc\text{-}buffers}\ s)\ (\mathsf{memory\text{-}of}\ s\ x)\ (\mathsf{memory\text{-}of}\ s'\ x)) \wedge \\ \quad (\forall x.\ \mathsf{cdt\text{-}integrity}\ p\ x\ (\mathsf{cdt}\ s\ x,\ \mathsf{is\text{-}original\text{-}cap}\ s\ x)\ (\mathsf{cdt}\ s'\ x,\ \mathsf{is\text{-}original\text{-}cap}\ s' \\ \quad x))\end{array}$$

This says that a transition from s to s' satisfies access control integrity if all kernel objects in the kernel object heap (kheap s), user memory and the capability derivation tree (cdt s) were changed in an acceptable manner.

The object level integrity predicate is defined by eight introduction rules. The following three rules are representative:

$$\frac{x = \mathsf{pasSubject}\ p}{\mathsf{object\text{-}integrity}\ p\ x\ ko\ ko'} \qquad \frac{ko = ko'}{\mathsf{object\text{-}integrity}\ p\ x\ ko\ ko'}$$

$$\frac{\begin{array}{c} ko = \mathsf{Some}\ (\mathsf{TCB}\ tcb) \qquad ko' = \mathsf{Some}\ (\mathsf{TCB}\ tcb') \\ \exists\, ctxt'.\ tcb' = tcb(\!|\mathsf{tcb\text{-}context} := ctxt',\ \mathsf{tcb\text{-}state} := \mathsf{Running}|\!) \\ \mathsf{receive\text{-}blocked\text{-}on}\ ep\ (\mathsf{tcb\text{-}state}\ tcb) \qquad \mathsf{auth} \in \{\mathsf{SyncSend},\ \mathsf{AsyncSend}\} \\ (\mathsf{pasSubject}\ p,\ \mathsf{auth},\ \mathsf{pasObjectAbs}\ p\ ep) \in \mathsf{pasPolicy}\ p \end{array}}{\mathsf{object\text{-}integrity}\ p\ l'\ ko\ ko'}$$

These cases allow the subject to make any change to itself, to leave anything unchanged, and to send a message through an endpoint it has send access to and into the registers (called the tcb-context here) of a waiting receiver. Note that it is guaranteed that the receiver's registers are changed only if the message transfer completes and that the receiver's state is changed to Running. It is likewise guaranteed by memory-integrity that a receiver's in-memory message buffer is changed only if the message transfer completes, which is the reason for the complexity of the arguments of the memory-integrity predicate above.

The additional object-integrity cases include a broadly symmetric case for receiving a message and resuming the sender, for resetting an endpoint and evicting a waiting sender or receiver, for updating the list of threads waiting at an endpoint, and for removing a virtual address space the subject owns from the active virtual address space database.

The cdt-integrity predicate limits all cdt changes to the subject's label.

The crucial property about integrity is transitivity:

Lemma 1. integrity $p\ s_0\ s_1 \wedge$ integrity $p\ s_1\ s_2 \longrightarrow$ integrity $p\ s_0\ s_2$

This must be true at the top level for our statement about a single system call to compose over an execution which is a sequence of such calls.

Integrity is also trivially reflexive:

Lemma 2. integrity $p\ s\ s$

3.5 Top Level Statements

Both integrity and pas-refined should be invariants of the system, which can be demonstrated using Hoare triples in a framework for reasoning about state monads in Isabelle/HOL. We have previously reported on this framework in depth [5]. In summary, the precondition of a Hoare triple in this framework is a predicate on the pre-state, the post condition is a predicate on the return value of the function and the post-state. In the case below, the return value is *unit* and can be ignored in the post condition. The framework provides a definition, logic, and automation for assembling such triples.

Theorem 1 (Integrity). *The property* integrity *pas st holds after all kernel calls, assuming that the protection state refines the policy, assuming the general system invariants* invs *and* ct-active *(current thread is active) for non-interrupt events, assuming the policy subject is the current thread, and assuming that the kernel state st is the state at the beginning of the kernel call. In Isabelle:*

$$\begin{array}{l}\{\!|\mathsf{pas\text{-}refined}\ pas \cap \mathsf{invs} \cap (\lambda s.\ ev \neq \mathsf{Interrupt} \longrightarrow \mathsf{ct\text{-}active}\ s) \cap \\ \ \mathsf{is\text{-}subject}\ pas \circ \mathsf{cur\text{-}thread} \cap (\lambda s.\ s = st)|\!\} \\ \mathsf{call\text{-}kernel}\ ev \\ \{\!|\lambda\text{-}.\ \mathsf{integrity}\ pas\ st|\!\}\end{array}$$

We have shown in previous work [12] that the preconditions invs and $ev \neq$ Interrupt $\longrightarrow$ ct-active s hold for any system execution at kernel entry.

Theorem 2 (Authority Confinement). *The property* pas-refined *pas is invariant over kernel calls, assuming again the general system invariants* invs *and* ct-active *for non-interrupt events, and assuming that the current subject of the policy is the current thread.*

$$\begin{array}{l}\{\!|\mathsf{pas\text{-}refined}\ pas \cap \mathsf{invs} \cap (\lambda s.\ ev \neq \mathsf{Interrupt} \longrightarrow \mathsf{ct\text{-}active}\ s) \cap \\ \ \mathsf{is\text{-}subject}\ pas \circ \mathsf{cur\text{-}thread}|\!\} \\ \mathsf{call\text{-}kernel}\ ev \\ \{\!|\lambda\text{-}.\ \mathsf{pas\text{-}refined}\ pas|\!\}\end{array}$$

We discuss the proof of these two theorems in the next section.

Via the refinement proof shown in previous work [12], both of these properties transfer to the C code level of the kernel. The guarantees that integrity *pas st* makes about user memory transfer directly, but the guarantees pas-refined *pas* and integrity *pas st* make about kernel-private state are mapped to their image across the refinement relation, which means we may lose some precision. The protection state of the kernel maps across the refinement relation precisely, the only difference between the model-level and C-level capability types being encoding.

The remainder of the system state does not translate so simply, but we contend that this does not matter. We envision the integrity theorem being useful mainly as a framing rule, with a component programmer appealing to the integrity theorem to exclude interference from other components and to the kernel model to reason about the component's own actions. In this case the programmer is interested not in the precise C state of the private kernel data, but about the related kernel model state. The integrity theorem will then provide exactly what is needed.

For proving confidentiality in the future, we may have to be more careful, because abstraction may hide sources of information that exist in the C system.

4 Proof and Application

4.1 Proof

The bulk of the proof effort was showing two Hoare triples for each kernel function: one to prove pas-refined as a postcondition, and one to prove integrity. These lemmas are convenient to use within the Hoare framework as we can phrase them in a predicate preservation style. In the case of the integrity predicate, we use a form of the Hoare triple (the left hand side of the following equality) which encapsulates transitivity and is easy to compose sequentially. This form is equivalent to the more explicit form (the right hand side) as a consequence of reflexivity and transitivity:

$$\begin{aligned}&\forall P.\ (\forall st.\ \{\!|\lambda s.\ \mathsf{integrity}\ pas\ st\ s \wedge P\ s|\!\}\ f\ \{\!|\lambda rv.\ \mathsf{integrity}\ pas\ st|\!\}) = \\ &\qquad (\forall st.\ \{\!|\lambda s.\ s = st \wedge P\ s|\!\}\ f\ \{\!|\lambda rv.\ \mathsf{integrity}\ pas\ st|\!\})\end{aligned}$$

The proof was accomplished by working through the kernel's call graph from bottom to top, establishing appropriate preconditions for confinement and integrity for each function. Some appeal was made to previously proven invariants.

The proof effort generally proceeded smoothly because of the strength of the abstraction we are making. We allow the subject to change arbitrarily anything with its label, we map most kinds of access to the Control authority, and we require anything to which the subject has Control authority to share the subject's label. These broad brushstroke justifications are easy to apply, and were valid for many code paths of the kernel.

For example, CSpace updates always occur within the subject. As preconditions for various functions such as cap-move and cap-insert we assert that the address of the CSpace node being updated has the subject's label and, for pas-refined preservation, that all authority contained in the new capabilities being added is possessed by the subject in the policy. These preconditions are properties about the static policy, not the dynamic current state, which makes them easy to propagate through the proof.

The concept of a static policy gave us further advantages. By comparison, we had previously attempted two different variations on a proof that seL4 directly refines the take-grant security model [15]. These proof attempts were mired in difficulties, because seL4 execution steps are larger than take-grant steps. In between the take-grant steps of a single seL4 kernel call, their preconditions may be violated because capabilities may have moved or disappeared, and so the steps could not be composed easily. This seemed particularly unfortunate considering that the take-grant authority model has a known static supremum. Comparing against this static graph instead yields something like our current approach.

Another advantage we have in this proof effort is the existing abstraction from the C code up to our kernel model. The cdt (capability derivation tree) and endpoint queues described already must be implemented in C through pointer linked datastructures. In C the cdt is encoded in prefix order as a linked list. When a subject manipulates its own capabilities it may cause updates to pointers in list-adjacent capabilities it does not control. In the abstract kernel model the cdt is represented as a partial map from child to parent nodes, making all of our subject's CSpace operations local. It would be possible to phrase an integrity predicate on the C level which allowed appropriate pointer updates, but we think it would be extremely difficult to work with.

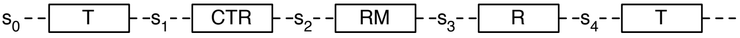
The combined proof scripts for the two access control properties, including the definitions of all formalisms and the SAC example, comprise 10500 lines of Isabelle/HOL source. The proof was completed over a 4 month period and consumed about 10 person months of effort. Relative to other proofs about the kernel model this was rapid progress. Modifications to this proof have also been fast, with the addition of the cdt-integrity aspect of the integrity property being finished in a single day.

During the proof we did not find any behaviour in seL4 that would break the integrity property nor did we need to change the specification or the code of seL4. We did encounter and clarify known unwanted API complexity which is expressed in our policy wellformedness assumption. One such known problem is that the API optimisation for invoking the equivalent of a remote procedure call in a server thread confers so much authority to the client that they have to reside in the same policy label. This means the optimisation cannot be used between trust boundaries. An alternative design was already scheduled, but is not implemented yet.

We have found a small number of access control violations in seL4's specification, but we found them during a previous proof before this work began. These problems related to capability rights that made little sense, such as read-only Thread capabilities, and had not been well examined. The problem was solved by purging these rights.

4.2 Application

The application scenario of the integrity theorem is a system comprising trusted as well as untrusted components such as in the SAC example. Such scenarios are problematic for purely mandatory access control systems such as subsystems in a take-grant setting [15,7], because the trusted component typically needs to be given too much authority for access control alone to enforce the system's security goals (hence the need for trust). Our formulation of integrity enforcement per subject provides more flexibility. Consider a sample trace of kernel executions on behalf of various components in the SAC.



The example policy in Fig. 3 satisfies our wellformedness condition for the components R, T, and CTR. It does not satisfy wellformedness for RM. This means, given the initial state s_0, we can predict bounds for authority and state mutation using our theorems up to s_2 before RM runs. Since RM is the trusted component, we do not apply the integrity theorem, but reason about its (known) behaviour instead and get a precise characterisation of s_3. From here on, the integrity theorem applies again, and so on. Suitably constructed, the bounds will be low enough to enforce a system wide state invariant over all possible traces which in turn, if chosen appropriately, will imply the security goal.

As hinted at in Sect. 2, at state s_2, the RM component could use its excessive authority to reconfigure the system such that a new R is now connected to networks A and D. This setup would be incompatible with the policy applied to the transitions before, but a new policy reflecting the reconfiguration can be constructed that applies from there on. If that new policy and the reconfiguration are compatible with the system invariant we can conclude the security goal for a system that dynamically changes its high-level policy even if the integrity theorem itself assumes a static policy per kernel event.

For reasoning about such systems, it is convenient to lift wellformedness and therefore pas-refined to sets of allowed subjects instead of a single current actor. This means the same instantiation of the theorem can be applied and chained over traces without further proof. We have formulated and proved the lifted version, but omit the details here. They add no further insight.

The set versions of integrity and pas-refined are also useful in a more restricted, but common scenario where the kernel is employed as a pure separation kernel or hypervisor. In this scenario, all components would be considered untrusted, and there can be one system-wide policy that is wellformed for the set of all components. Wellformedness can be shown once per system and the subjective integrity theorem collapses to a traditional access control formulation of integrity that applies to all subjects.

4.3 Limitations

The limitations of the theorem as presented mostly reflect API complexities that we circumvented by making assumptions on the policy.

The strongest such assumption is that we require two components with a Grant connection to map to the same label. This is no more than what a traditional take-grant analysis amounts to [7], but in our subjective setting there would be no strong reason to forbid Grant from trusted to untrusted components if the authority transmitted is within policy bounds. The difficulties with this and with interrupt delivery were discussed in Sect. 3.2. Trusted components can still delegate capabilities via shared CNodes, which appear in our graph as Control edges. This is the approach taken by the RM component of the SAC.

Another limitation is that the theorem provides relatively coarse bounds. This is required to achieve the level of abstraction we are seeking, but it is imaginable that the bounds could be too high for a particular frame condition that is required. In this case, one could always fall back to reasoning about the precise kernel behaviour for the event under consideration, but it was of course the purpose of the theorem to be able to avoid this.

5 Related Work

Security properties have been the goal for OS verification from the beginning: the first projects in this space UCLA Secure Unix [19] and PSOS [8] (Provably Secure OS) were already aiming at such proofs. For a general overview on OS verification, we refer to Klein [11] and concentrate below on security in particular.

A range of security properties have been proven of OS kernels and their access control systems in the past, such as Guttman et al's [9] work on information flow, or Krohn et al [13] on non-interference. These proofs work on higher-level kernel abstractions. Closest to our level of fidelity comes Richards [17] in his description of the security analysis of the INTEGRITY-178B kernel in ACL2. Even this model is still connected to source code manually.

As mentioned, seL4 implements a variant of the take-grant capability system [15], a key property of which is the transitive, reflexive, and symmetric closure over all Grant connections. This closure provides an authority bound and is invariant over system execution. Similar properties hold for a broader class, such as general object-capability systems [16].

We have previously proved in Isabelle that the take-grant bound holds for an abstraction of the seL4 API [7,4]. The EROS kernel supports a similar model with similar proof [18]. However, these abstractions were simpler than the one presented here and not formally connected to code.

Compared to pure take-grant, our subjective formulation of integrity is less pessimistic: it allows certain trusted components, which are separately verified, to possess sufficient Control rights to propagate authority beyond that allowed by the policy.

6 Conclusions

In this paper, we have presented the first formal proof of integrity enforcement and authority confinement for a general-purpose OS kernel implementation. These properties together provide a powerful framing condition for the verification of whole systems constructed on top of seL4, which we have argued with reference to a case study on a real example system.

We have shown that the real-life complexity in reasoning about a general-purpose kernel implementation can be managed using abstraction. In particular, our formalisation avoids direct reasoning about the protection state, which can change over time, by representing it via a separate policy abstraction that is constant across system calls. Integrity asserts that state mutations must be permitted by this policy, while authority confinement asserts that the protection state cannot evolve to contradict the policy.

This work clearly demonstrates that proving high-level security properties of real kernel implementations, and the systems they host, is now a reality. We should demand nothing less for security-critical applications in the future.

Acknowledgements. We thank Magnus Myreen for commenting on a draft of this paper.

References

1. Andronick, J., Bourke, T., Derrin, P., Elphinstone, K., Greenaway, D., Klein, G., Kolanski, R., Sewell, T., Winwood, S.: Abstract formal specification of the seL4/ARMv6 API (January 2011), `http://ertos.nicta.com.au/software/seL4/`
2. Andronick, J., Greenaway, D., Elphinstone, K.: Towards proving security in the presence of large untrusted components. In: Klein, G., Huuck, R., Schlich, B. (eds.) 5th SSV. USENIX, Vancouver (2010)
3. Bell, D., LaPadula, L.: Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp (March 1976)
4. Boyton, A.: A verified shared capability model. In: Klein, G., Huuck, R., Schlich, B. (eds.) 4th SSV. ENTCS, vol. 254, pp. 25–44. Elsevier, Amsterdam (2009)
5. 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)
6. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. CACM 9, 143–155 (1966)
7. Elkaduwe, D., Klein, G., Elphinstone, K.: Verified protection model of the seL4 microkernel. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 99–114. Springer, Heidelberg (2008)
8. Feiertag, R.J., Neumann, P.G.: The foundations of a provably secure operating system (PSOS). In: AFIPS Conf. Proc., 1979 National Comp. Conf., New York, NY, USA, pp. 329–334 (June 1979)
9. Guttman, J., Herzog, A., Ramsdell, J., Skorupka, C.: Verifying information flow goals in security-enhanced linux. J. Comp. Security 13, 115–134 (2005)
10. Jaeger, T.: Operating System Security. Morgan & Claypool Publishers, San Francisco (2008)
11. Klein, G.: Operating system verification—an overview. Sādhanā 34(1), 27–69 (2009)
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSP, pp. 207–220. ACM, Big Sky (2009)
13. Krohn, M., Tromer, E.: Noninterference for a practical DIFC-based operating system. In: IEEE Symp. Security & Privacy, pp. 61–76 (2009)
14. Lampson, B.W.: Protection. In: 5th Princeton Symposium on Information Sciences and Systems, pp. 437–443. Princeton University, Princeton (1971); Reprinted in Operat. Syst. Rev. 8(1), 18–24 (1974)
15. Lipton, R.J., Snyder, L.: A linear time algorithm for deciding subject security. J. ACM 24(3), 455–464 (1977)
16. Murray, T., Lowe, G.: Analysing the information flow properties of object-capability patterns. In: Degano, P., Guttman, J.D. (eds.) FAST 2009. LNCS, vol. 5983, pp. 81–95. Springer, Heidelberg (2010)
17. Richards, R.J.: Modeling and security analysis of a commercial real-time operating system kernel. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 301–322. Springer, Heidelberg (2010)
18. Shapiro, J.S., Weber, S.: Verifying the EROS confinement mechanism. In: IEEE Symp. Security & Privacy, Washington, DC, USA, pp. 166–181 (May 2000)
19. Walker, B.J., Kemmerer, R.A., Popek, G.J.: Specification and verification of the UCLA Unix security kernel. CACM 23(2), 118–131 (1980)

A Formalisation of the Myhill-Nerode Theorem Based on Regular Expressions (Proof Pearl)

Chunhan Wu[1], Xingyuan Zhang[1], and Christian Urban[2]

[1] PLA University of Science and Technology, China
[2] TU Munich, Germany

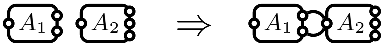
Abstract. There are numerous textbooks on regular languages. Nearly all of them introduce the subject by describing finite automata and only mentioning on the side a connection with regular expressions. Unfortunately, automata are difficult to formalise in HOL-based theorem provers. The reason is that they need to be represented as graphs, matrices or functions, none of which are inductive datatypes. Also convenient operations for disjoint unions of graphs and functions are not easily formalisiable in HOL. In contrast, regular expressions can be defined conveniently as a datatype and a corresponding reasoning infrastructure comes for free. We show in this paper that a central result from formal language theory—the Myhill-Nerode theorem—can be recreated using only regular expressions.

1 Introduction

Regular languages are an important and well-understood subject in Computer Science, with many beautiful theorems and many useful algorithms. There is a wide range of textbooks on this subject, many of which are aimed at students and contain very detailed 'pencil-and-paper' proofs (e.g. [7]). It seems natural to exercise theorem provers by formalising the theorems and by verifying formally the algorithms.

There is however a problem: the typical approach to regular languages is to introduce finite automata and then define everything in terms of them. For example, a regular language is normally defined as one whose strings are recognised by a finite deterministic automaton. This approach has many benefits. Among them is the fact that it is easy to convince oneself that regular languages are closed under complementation: one just has to exchange the accepting and non-accepting states in the corresponding automaton to obtain an automaton for the complement language. The problem, however, lies with formalising such reasoning in a HOL-based theorem prover, in our case Isabelle/HOL. Automata are built up from states and transitions that need to be represented as graphs, matrices or functions, none of which can be defined as an inductive datatype.

In case of graphs and matrices, this means we have to build our own reasoning infrastructure for them, as neither Isabelle/HOL nor HOL4 nor HOLlight support them with libraries. Even worse, reasoning about graphs and matrices can be a real hassle in HOL-based theorem provers. Consider for example the operation of sequencing two automata, say A_1 and A_2, by connecting the accepting states of A_1 to the initial state of A_2:



M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 341–356, 2011.

On 'paper' we can define the corresponding graph in terms of the disjoint union of the state nodes. Unfortunately in HOL, the standard definition for disjoint union, namely

$$A_1 \uplus A_2 \stackrel{def}{=} \{(\mathit{1}, x) \mid x \in A_1\} \cup \{(2, y) \mid y \in A_2\} \tag{1}$$

changes the type—the disjoint union is not a set, but a set of pairs. Using this definition for disjoint union means we do not have a single type for automata and hence will not be able to state certain properties about *all* automata, since there is no type quantification available in HOL (unlike in Coq, for example). An alternative, which provides us with a single type for automata, is to give every state node an identity, for example a natural number, and then be careful to rename these identities apart whenever connecting two automata. This results in clunky proofs establishing that properties are invariant under renaming. Similarly, connecting two automata represented as matrices results in very adhoc constructions, which are not pleasant to reason about.

Functions are much better supported in Isabelle/HOL, but they still lead to similar problems as with graphs. Composing, for example, two non-deterministic automata in parallel requires also the formalisation of disjoint unions. Nipkow [9] dismisses for this the option of using identities, because it leads according to him to "messy proofs". He opts for a variant of (1) using bit lists, but writes

> *"All lemmas appear obvious given a picture of the composition of automata... Yet their proofs require a painful amount of detail."*

and

> *"If the reader finds the above treatment in terms of bit lists revoltingly concrete, I cannot disagree. A more abstract approach is clearly desirable."*

Moreover, it is not so clear how to conveniently impose a finiteness condition upon functions in order to represent *finite* automata. The best is probably to resort to more advanced reasoning frameworks, such as *locales* or *type classes*, which are *not* available in all HOL-based theorem provers.

Because of these problems to do with representing automata, there seems to be no substantial formalisation of automata theory and regular languages carried out in HOL-based theorem provers. Nipkow [9] establishes the link between regular expressions and automata in the context of lexing. Berghofer and Reiter [2] formalise automata working over bit strings in the context of Presburger arithmetic. The only larger formalisations of automata theory are carried out in Nuprl [4] and in Coq [5].

In this paper, we will not attempt to formalise automata theory in Isabelle/HOL, but take a different approach to regular languages. Instead of defining a regular language as one where there exists an automaton that recognises all strings of the language, we define a regular language as:

Definition 1. *A language A is* regular*, provided there is a regular expression that matches all strings of A.*

The reason is that regular expressions, unlike graphs, matrices and functions, can be easily defined as inductive datatype. Consequently a corresponding reasoning infrastructure comes for free. This has recently been exploited in HOL4 with a formalisation of regular expression matching based on derivatives [11] and with an equivalence checker for regular expressions in Isabelle/HOL [8]. The purpose of this paper is to show that a central result about regular languages—the Myhill-Nerode theorem—can be recreated by only using regular expressions. This theorem gives necessary and sufficient conditions for when a language is regular. As a corollary of this theorem we can easily establish the usual closure properties, including complementation, for regular languages.

Contributions: There is an extensive literature on regular languages. To our best knowledge, our proof of the Myhill-Nerode theorem is the first that is based on regular expressions, only. We prove the part of this theorem stating that a regular expression has only finitely many partitions using certain tagging-functions. Again to our best knowledge, these tagging-functions have not been used before to establish the Myhill-Nerode theorem.

2 Preliminaries

Strings in Isabelle/HOL are lists of characters with the *empty string* being represented by the empty list, written $[]$. *Languages* are sets of strings. The language containing all strings is written in Isabelle/HOL as *UNIV*. The concatenation of two languages is written $A \cdot B$ and a language raised to the power n is written A^n. They are defined as usual

$$A \cdot B \overset{def}{=} \{s_1 \mathbin{@} s_2 \mid s_1 \in A \land s_2 \in B\} \qquad A^0 \overset{def}{=} \{[]\} \qquad A^{n+1} \overset{def}{=} A \cdot A^n$$

where @ is the list-append operation. The Kleene-star of a language A is defined as the union over all powers, namely $A^\star \overset{def}{=} \bigcup_n A^n$. In the paper we will make use of the following properties of these constructions.

Proposition 1.
(i) $A^\star = \{[]\} \cup A \cdot A^\star$
(ii) If $[] \notin A$ *and* $s \in A^{n+1}$ *then* $n < |s|$.
(iii) $B \cdot (\bigcup_n A^n) = (\bigcup_n B \cdot A^n)$

In (ii) we use the notation $|s|$ for the length of a string; this property states that if $[] \notin A$ then the lengths of the strings in A^{n+1} must be longer than n. We omit the proofs for these properties, but invite the reader to consult our formalisation.[1]

The notation in Isabelle/HOL for the quotient of a language A according to an equivalence relation $\approx$ is $A /\!\!/ \approx$. We will write $[\![x]\!]_\approx$ for the equivalence class defined as $\{y \mid y \approx x\}$.

Central to our proof will be the solution of equational systems involving equivalence classes of languages. For this we will use Arden's Lemma [3], which solves equations of the form $X = A \cdot X \cup B$ provided $[] \notin A$. However we will need the following 'reverse' version of Arden's Lemma ('reverse' in the sense of changing the order of $A \cdot X$ to $X \cdot A$).

Lemma 1 (Reverse Arden's Lemma)
If $[] \notin A$ *then* $X = X \cdot A \cup B$ *if and only if* $X = B \cdot A^\star$.

[1] Available at http://www4.in.tum.de/~urbanc/regexp.html

Proof For the right-to-left direction we assume $X = B \cdot A^\star$ and show that $X = X \cdot A \cup B$ holds. From Prop. $1(i)$ we have $A^\star = \{[]\} \cup A \cdot A^\star$, which is equal to $A^\star = \{[]\} \cup A^\star \cdot A$. Adding B to both sides gives $B \cdot A^\star = B \cdot (\{[]\} \cup A^\star \cdot A)$, whose right-hand side is equal to $(B \cdot A^\star) \cdot A \cup B$. This completes this direction.

For the other direction we assume $X = X \cdot A \cup B$. By a simple induction on n, we can establish the property

$$(*) \qquad X = X \cdot A^{n+1} \cup (\bigcup_{m \in \{0..n\}} B \cdot A^m)$$

Using this property we can show that $B \cdot A^n \subseteq X$ holds for all n. From this we can infer $B \cdot A^\star \subseteq X$ using the definition of $\star$. For the inclusion in the other direction we assume a string s with length k is an element in X. Since $[] \notin A$ we know by Prop. $1(ii)$ that $s \notin X \cdot A^{k+1}$ since its length is only k (the strings in $X \cdot A^{k+1}$ are all longer). From $(*)$ it follows then that s must be an element in $\bigcup_{m \in \{0..k\}} B \cdot A^m$. This in turn implies that s is in $\bigcup_n B \cdot A^n$. Using Prop. $1(iii)$ this is equal to $B \cdot A^\star$, as we needed to show. □

Regular expressions are defined as the inductive datatype

$$r ::= NULL \mid EMPTY \mid CHAR\ c \mid SEQ\ r\ r \mid ALT\ r\ r \mid STAR\ r$$

and the language matched by a regular expression is defined as

$$\begin{array}{rclcrcl}
\mathcal{L}(NULL) & \stackrel{def}{=} & \varnothing & \qquad & \mathcal{L}(SEQ\ r_1\ r_2) & \stackrel{def}{=} & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(EMPTY) & \stackrel{def}{=} & \{[]\} & & \mathcal{L}(ALT\ r_1\ r_2) & \stackrel{def}{=} & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(CHAR\ c) & \stackrel{def}{=} & \{[c]\} & & \mathcal{L}(STAR\ r) & \stackrel{def}{=} & \mathcal{L}(r)^\star
\end{array}$$

Given a finite set of regular expressions rs, we will make use of the operation of generating a regular expression that matches the union of all languages of rs. We only need to know the existence of such a regular expression and therefore we use Isabelle/HOL's *fold_graph* and Hilbert's ε to define $+rs$. This operation, roughly speaking, folds ALT over the set rs with $NULL$ for the empty set. We can prove that for a finite set rs

$$\mathcal{L}(+rs) = \bigcup\ (\mathcal{L}\ `\ rs) \tag{2}$$

holds, whereby $\mathcal{L}\ `\ rs$ stands for the image of the set rs under function $\mathcal{L}$.

3 The Myhill-Nerode Theorem, First Part

The key definition in the Myhill-Nerode theorem is the *Myhill-Nerode relation*, which states that w.r.t. a language two strings are related, provided there is no distinguishing extension in this language. This can be defined as a tertiary relation.

Definition 2 (Myhill-Nerode Relation). *Given a language A, two strings x and y are Myhill-Nerode related provided*

$$x \approx_A y \stackrel{def}{=} \forall z.\ (x @ z \in A) = (y @ z \in A)$$

It is easy to see that $\approx_A$ is an equivalence relation, which partitions the set of all strings, *UNIV*, into a set of disjoint equivalence classes. To illustrate this quotient construction, let us give a simple example: consider the regular language containing just the string $[c]$. The relation $\approx_{\{[c]\}}$ partitions *UNIV* into three equivalence classes X_1, X_2 and X_3 as follows

$$X_1 = \{[]\} \qquad X_2 = \{[c]\} \qquad X_3 = \mathit{UNIV} - \{[], [c]\}$$

One direction of the Myhill-Nerode theorem establishes that if there are finitely many equivalence classes, like in the example above, then the language is regular. In our setting we therefore have to show:

Theorem 1. *If finite* $(\mathit{UNIV} /\!/ \approx_A)$ *then* $\exists\, r.\, A = \mathcal{L}(r)$.

To prove this theorem, we first define the set *finals A* as those equivalence classes from $\mathit{UNIV} /\!/ \approx_A$ that contain strings of A, namely

$$\mathit{finals}\, A \overset{def}{=} \{[\![s]\!]_{\approx_A} \mid s \in A\} \tag{3}$$

In our running example, X_2 is the only equivalence class in *finals* $\{[c]\}$. It is straightforward to show that in general $A = \bigcup \mathit{finals}\, A$ and $\mathit{finals}\, A \subseteq \mathit{UNIV} /\!/ \approx_A$ hold. Therefore if we know that there exists a regular expression for every equivalence class in *finals A* (which by assumption must be a finite set), then we can use $+$ to obtain a regular expression that matches every string in A.

Our proof of Thm. 1 relies on a method that can calculate a regular expression for *every* equivalence class, not just the ones in *finals A*. We first define the notion of *one-character-transition* between two equivalence classes

$$Y \overset{c}{\Longmapsto} X \overset{def}{=} Y \cdot \{[c]\} \subseteq X \tag{4}$$

which means that if we concatenate the character c to the end of all strings in the equivalence class Y, we obtain a subset of X. Note that we do not define an automaton here, we merely relate two sets (with the help of a character). In our concrete example we have $X_1 \overset{c}{\Longmapsto} X_2$, $X_1 \overset{d}{\Longmapsto} X_3$ with d being any other character than c, and $X_3 \overset{d}{\Longmapsto} X_3$ for any d.

Next we construct an *initial equational system* that contains an equation for each equivalence class. We first give an informal description of this construction. Suppose we have the equivalence classes X_1,...,X_n, there must be one and only one that contains the empty string [] (since equivalence classes are disjoint). Let us assume $[] \in X_1$. We build the following equational system

$$\begin{aligned}
X_1 &= (Y_{11}, \mathit{CHAR}\, c_{11}) + \ldots + (Y_{1p}, \mathit{CHAR}\, c_{1p}) + \lambda(\mathit{EMPTY}) \\
X_2 &= (Y_{21}, \mathit{CHAR}\, c_{21}) + \ldots + (Y_{2o}, \mathit{CHAR}\, c_{2o}) \\
&\ \ \vdots \\
X_n &= (Y_{n1}, \mathit{CHAR}\, c_{n1}) + \ldots + (Y_{nq}, \mathit{CHAR}\, c_{nq})
\end{aligned}$$

where the terms $(Y_{ij}, \mathit{CHAR}\ c_{ij})$ stand for all transitions $Y_{ij} \overset{c_{ij}}{\Longmapsto} X_i$. There can only be finitely many terms of the form $(Y_{ij}, \mathit{CHAR}\ c_{ij})$ in a right-hand side since by assumption there are only finitely many equivalence classes and only finitely many characters. The term $\lambda(\mathit{EMPTY})$ in the first equation acts as a marker for the initial state, that is the equivalence class containing [].[2] Overloading the function $\mathcal{L}$ for the two kinds of terms in the equational system, we have

$$\mathcal{L}(Y, r) \overset{\mathit{def}}{=} Y \cdot \mathcal{L}(r) \qquad\qquad \mathcal{L}(\lambda(r)) \overset{\mathit{def}}{=} \mathcal{L}(r)$$

and we can prove for $X_{2..n}$ that the following equations

$$X_i = \mathcal{L}(Y_{i1}, \mathit{CHAR}\ c_{i1}) \cup \ldots \cup \mathcal{L}(Y_{iq}, \mathit{CHAR}\ c_{iq}). \tag{5}$$

hold. Similarly for X_1 we can show the following equation

$$X_1 = \mathcal{L}(Y_{11}, \mathit{CHAR}\ c_{11}) \cup \ldots \cup \mathcal{L}(Y_{1p}, \mathit{CHAR}\ c_{1p}) \cup \mathcal{L}(\lambda(\mathit{EMPTY})). \tag{6}$$

The reason for adding the λ-marker to our initial equational system is to obtain this equation: it only holds with the marker, since none of the other terms contain the empty string. The point of the initial equational system is that solving it means we will be able to extract a regular expression for every equivalence class.

Our representation for the equations in Isabelle/HOL are pairs, where the first component is an equivalence class (a set of strings) and the second component is a set of terms. Given a set of equivalence classes *CS*, our initial equational system *Init CS* is thus formally defined as

$$\begin{array}{rcl} \mathit{Init\_rhs}\ \mathit{CS}\ X & \overset{\mathit{def}}{=} & \mathit{if}\ [] \in X \\ & & \mathit{then}\ \{(Y, \mathit{CHAR}\ c) \mid Y \in \mathit{CS} \wedge Y \overset{c}{\Longmapsto} X\} \cup \{\lambda(\mathit{EMPTY})\} \\ & & \mathit{else}\ \{(Y, \mathit{CHAR}\ c) \mid Y \in \mathit{CS} \wedge Y \overset{c}{\Longmapsto} X\} \\ \mathit{Init}\ \mathit{CS} & \overset{\mathit{def}}{=} & \{(X, \mathit{Init\_rhs}\ \mathit{CS}\ X) \mid X \in \mathit{CS}\} \end{array} \tag{7}$$

Because we use sets of terms for representing the right-hand sides of equations, we can prove (5) and (6) more concisely as

Lemma 2. *If* $(X, \mathit{rhs}) \in \mathit{Init}\ (\mathit{UNIV} /\!/ {\approx_A})$ *then* $X = \bigcup \mathcal{L}\ `\ \mathit{rhs}$.

Our proof of Thm. 1 will proceed by transforming the initial equational system into one in *solved form* maintaining the invariant in Lem. 2. From the solved form we will be able to read off the regular expressions.

In order to transform an equational system into solved form, we have two operations: one that takes an equation of the form $X = \mathit{rhs}$ and removes any recursive occurrences of

[2] Note that we mark, roughly speaking, the single 'initial' state in the equational system, which is different from the method by Brzozowski [3], where he marks the 'terminal' states. We are forced to set up the equational system in our way, because the Myhill-Nerode relation determines the 'direction' of the transitions—the successor 'state' of an equivalence class *Y* can be reached by adding a character to the end of *Y*. This is also the reason why we have to use our reverse version of Arden's Lemma.

X in the *rhs* using our variant of Arden's Lemma. The other operation takes an equation $X = rhs$ and substitutes *X* throughout the rest of the equational system adjusting the remaining regular expressions appropriately. To define this adjustment we define the *append-operation* taking a term and a regular expression as argument

$$(Y, r_2) \lhd r_1 \overset{def}{=} (Y, SEQ\, r_2\, r_1) \qquad \lambda(r_2) \lhd r_1 \overset{def}{=} \lambda(SEQ\, r_2\, r_1)$$

We lift this operation to entire right-hand sides of equations, written as $rhs \lhd r$. With this we can define the *arden-operation* for an equation of the form $X = rhs$ as:

$$\begin{array}{rl} Arden\; X\; rhs \overset{def}{=} & let \\ & \quad rhs' = rhs - \{(X, r) \mid (X, r) \in rhs\} \\ & \quad\;\; r' = STAR\; (+\{r \mid (X, r) \in rhs\}) \\ & in\;\; rhs' \lhd r' \end{array} \tag{8}$$

In this definition, we first delete all terms of the form (X, r) from *rhs*; then we calculate the combined regular expressions for all *r* coming from the deleted (X, r), and take the *STAR* of it; finally we append this regular expression to rhs'. It can be easily seen that this operation mimics Arden's Lemma on the level of equations. To ensure the non-emptiness condition of Arden's Lemma we say that a right-hand side is *ardenable* provided

$$ardenable\; rhs \overset{def}{=} \forall\, Y\, r.\; (Y, r) \in rhs \longrightarrow [] \notin \mathcal{L}(r)$$

This allows us to prove a version of Arden's Lemma on the level of equations.

Lemma 3. *Given an equation* $X = rhs$. *If* $X = \bigcup \mathcal{L}$ ‘ *rhs*, *ardenable rhs*, *and finite rhs*, *then* $X = \bigcup \mathcal{L}$ ‘ $(Arden\; X\; rhs)$.

Our *ardenable* condition is slightly stronger than needed for applying Arden's Lemma, but we can still ensure that it holds troughout our algorithm of transforming equations into solved form. The *substitution-operation* takes an equation of the form $X = xrhs$ and substitutes it into the right-hand side *rhs*.

$$\begin{array}{rl} Subst\; rhs\; X\; xrhs \overset{def}{=} & let \\ & \quad rhs' = rhs - \{(X, r) \mid (X, r) \in rhs\} \\ & \quad\;\; r' = +\{r \mid (X, r) \in rhs\} \\ & in\;\; rhs' \cup (xrhs \lhd r') \end{array}$$

We again delete first all occurrences of (X, r) in *rhs*; we then calculate the regular expression corresponding to the deleted terms; finally we append this regular expression to *xrhs* and union it up with rhs'. When we use the substitution operation we will arrange it so that *xrhs* does not contain any occurrence of *X*.

With these two operations in place, we can define the operation that removes one equation from an equational systems *ES*. The operation *Subst_all* substitutes an equation $X = xrhs$ throughout an equational system *ES*; *Remove* then completely removes such an equation from *ES* by substituting it to the rest of the equational system, but first eliminating all recursive occurrences of *X* by applying *Arden* to *xrhs*.

$$\begin{aligned} \mathit{Subst\_all}\ \mathit{ES}\ X\ \mathit{xrhs} &\overset{\mathit{def}}{=} \{(Y, \mathit{Subst}\ \mathit{yrhs}\ X\ \mathit{xrhs}) \mid (Y, \mathit{yrhs}) \in \mathit{ES}\} \\ \mathit{Remove}\ \mathit{ES}\ X\ \mathit{xrhs} &\overset{\mathit{def}}{=} \mathit{Subst\_all}\ (\mathit{ES} - \{(X, \mathit{xrhs})\})\ X\ (\mathit{Arden}\ X\ \mathit{xrhs}) \end{aligned}$$

Finally, we can define how an equational system should be solved. For this we will need to iterate the process of eliminating equations until only one equation will be left in the system. However, we do not just want to have any equation as being the last one, but the one involving the equivalence class for which we want to calculate the regular expression. Let us suppose this equivalence class is X. Since X is the one to be solved, in every iteration step we have to pick an equation to be eliminated that is different from X. In this way X is kept to the final step. The choice is implemented using Hilbert's choice operator, written *SOME* in the definition below.

$$\begin{aligned} \mathit{Iter}\ X\ \mathit{ES} \overset{\mathit{def}}{=}\ & \mathit{let} \\ & \quad (Y, \mathit{yrhs}) = \mathit{SOME}\ (Y, \mathit{yrhs}).\ (Y, \mathit{yrhs}) \in \mathit{ES} \wedge X \neq Y \\ & \mathit{in}\ \ \mathit{Remove}\ \mathit{ES}\ Y\ \mathit{yrhs} \end{aligned}$$

The last definition we need applies *Iter* over and over until a condition *Cond* is *not* satisfied anymore. This condition states that there are more than one equation left in the equational system *ES*. To solve an equational system we use Isabelle/HOL's *while*-operator as follows:

$$\mathit{Solve}\ X\ \mathit{ES} \overset{\mathit{def}}{=} \mathit{while}\ \mathit{Cond}\ (\mathit{Iter}\ X)\ \mathit{ES}$$

We are not concerned here with the definition of this operator (see Berghofer and Nipkow [1]), but note that we eliminate in each *Iter*-step a single equation, and therefore have a well-founded termination order by taking the cardinality of the equational system *ES*. This enables us to prove properties about our definition of *Solve* when we 'call' it with the equivalence class X and the initial equational system $\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)$ from (7) using the principle:

$$\frac{\begin{array}{l} \mathit{invariant}\ (\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)) \\ \forall \mathit{ES}.\ \mathit{invariant}\ \mathit{ES} \wedge \mathit{Cond}\ \mathit{ES} \longrightarrow \mathit{invariant}\ (\mathit{Iter}\ X\ \mathit{ES}) \\ \forall \mathit{ES}.\ \mathit{invariant}\ \mathit{ES} \wedge \mathit{Cond}\ \mathit{ES} \longrightarrow \mathit{card}\ (\mathit{Iter}\ X\ \mathit{ES}) < \mathit{card}\ \mathit{ES} \\ \forall \mathit{ES}.\ \mathit{invariant}\ \mathit{ES} \wedge \neg\ \mathit{Cond}\ \mathit{ES} \longrightarrow P\ \mathit{ES} \end{array}}{P\ (\mathit{Solve}\ X\ (\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)))} \tag{9}$$

This principle states that given an invariant (which we will specify below) we can prove a property P involving *Solve*. For this we have to discharge the following proof obligations: first the initial equational system satisfies the invariant; second the iteration step *Iter* preserves the invariant as long as the condition *Cond* holds; third *Iter* decreases the termination order, and fourth that once the condition does not hold anymore then the property P must hold.

The property P in our proof will state that $\mathit{Solve}\ X\ (\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A))$ returns with a single equation $X = \mathit{xrhs}$ for some *xrhs*, and that this equational system still satisfies the invariant. In order to get the proof through, the invariant is composed of the following six properties:

$$\begin{array}{lll}
\textit{invariant ES} \overset{\textit{def}}{=} \textit{finite ES} & & (\textit{finiteness}) \\
\quad \wedge\ \forall\,(X, \textit{rhs}){\in}\textit{ES}.\,\textit{finite rhs} & & (\textit{finiteness rhs}) \\
\quad \wedge\ \forall\,(X, \textit{rhs}){\in}\textit{ES}.\ X = \bigcup \mathcal{L}\ `\ \textit{rhs} & & (\textit{soundness}) \\
\quad \wedge\ \forall X\ \textit{rhs}\ \textit{rhs}'.\ (X, \textit{rhs}) \in \textit{ES} \wedge (X, \textit{rhs}') \in \textit{ES} \longrightarrow \textit{rhs} = \textit{rhs}' & & \\
 & & (\textit{distinctness}) \\
\quad \wedge\ \forall\,(X, \textit{rhs}){\in}\textit{ES}.\ \textit{ardenable rhs} & & (\textit{ardenable}) \\
\quad \wedge\ \forall\,(X, \textit{rhs}){\in}\textit{ES}.\ \textit{rhss rhs} \subseteq \textit{lhss ES} & & (\textit{validity})
\end{array}$$

The first two ensure that the equational system is always finite (number of equations and number of terms in each equation); the second makes sure the 'meaning' of the equations is preserved under our transformations. The other properties are a bit more technical, but are needed to get our proof through. Distinctness states that every equation in the system is distinct. *Ardenable* ensures that we can always apply the *Arden* operation. The last property states that every *rhs* can only contain equivalence classes for which there is an equation. Therefore *lhss* is just the set containing the first components of an equational system, while *rhss* collects all equivalence classes X in the terms of the form (X, r). That means formally $\textit{lhss ES} \overset{\textit{def}}{=} \{X \mid (X, \textit{rhs}) \in \textit{ES}\}$ and $\textit{rhss rhs} \overset{\textit{def}}{=} \{X \mid (X, r) \in \textit{rhs}\}$.

It is straightforward to prove that the initial equational system satisfies the invariant.

Lemma 4. *If finite* $(\textit{UNIV}\,/\!/\!\approx_A)$ *then invariant* $(\textit{Init}\ (\textit{UNIV}\,/\!/\!\approx_A))$.

Proof. Finiteness is given by the assumption and the way how we set up the initial equational system. Soundness is proved in Lem. 2. Distinctness follows from the fact that the equivalence classes are disjoint. The *ardenable* property also follows from the setup of the initial equational system, as does validity. □

Next we show that *Iter* preserves the invariant.

Lemma 5. *If invariant ES and* $(X, \textit{rhs}) \in \textit{ES}$ *and Cond ES then invariant* $(\textit{Iter X ES})$.

Proof. The argument boils down to choosing an equation $Y = \textit{yrhs}$ to be eliminated and to show that $\textit{Subst\_all}\ (\textit{ES} - \{(Y, \textit{yrhs})\})\ Y\ (\textit{Arden}\ Y\, \textit{yrhs})$ preserves the invariant. We prove this as follows:

$$\forall\ \textit{ES}.\ \textit{invariant}\ (\textit{ES} \cup \{(Y, \textit{yrhs})\})\ \text{implies}\ \textit{invariant}\ (\textit{Subst\_all ES}\ Y\ (\textit{Arden}\ Y\, \textit{yrhs}))$$

Finiteness is straightforward, as the *Subst* and *Arden* operations keep the equational system finite. These operations also preserve soundness and distinctness (we proved soundness for *Arden* in Lem. 3). The property *ardenable* is clearly preserved because the append-operation cannot make a regular expression to match the empty string. Validity is given because *Arden* removes an equivalence class from *yrhs* and then *Subst_all* removes Y from the equational system. Having proved the implication above, we can instantiate *ES* with $\textit{ES} - \{(Y, \textit{yrhs})\}$ which matches with our proof-obligation of *Subst_all*. Since $\textit{ES} = \textit{ES} - \{(Y, \textit{yrhs})\} \cup \{(Y, \textit{yrhs})\}$, we can use the assumption to complete the proof. □

We also need the fact that *Iter* decreases the termination measure.

Lemma 6. *If invariant ES and* $(X,\ rhs) \in ES$ *and Cond ES then card* $(Iter\ X\ ES) <$ *card ES.*

Proof. By assumption we know that *ES* is finite and has more than one element. Therefore there must be an element $(Y,\ yrhs) \in ES$ with $(Y,\ yrhs) \neq (X,\ rhs)$. Using the distinctness property we can infer that $Y \neq X$. We further know that *Remove ES Y yrhs* removes the equation $Y = yrhs$ from the system, and therefore the cardinality of *Iter* strictly decreases. □

This brings us to our property we want to establish for *Solve*.

Lemma 7. *If finite* $(UNIV /\!\!/ \approx_A)$ *and* $X \in UNIV /\!\!/ \approx_A$ *then there exists a rhs such that Solve X* $(Init\ (UNIV /\!\!/ \approx_A)) = \{(X,\ rhs)\}$ *and invariant* $\{(X,\ rhs)\}$*.*

Proof. In order to prove this lemma using (9), we have to use a slightly stronger invariant since Lem. 5 and 6 have the precondition that $(X,\ rhs) \in ES$ for some *rhs*. This precondition is needed in order to choose in the *Iter*-step an equation that is not $X = rhs$. Therefore our invariant cannot be just *invariant ES*, but must be *invariant ES* $\wedge$ $(\exists\, rhs.\ (X,\ rhs) \in ES)$. By assumption $X \in UNIV /\!\!/ \approx_A$ and Lem. 4, the more general invariant holds for the initial equational system. This is premise 1 of (9). Premise 2 is given by Lem. 5 and the fact that *Iter* might modify the *rhs* in the equation $X = rhs$, but does not remove it. Premise 3 of (9) is by Lem. 6. Now in premise 4 we like to show that there exists a *rhs* such that $ES = \{(X,\ rhs)\}$ and that *invariant* $\{(X,\ rhs)\}$ holds, provided the condition *Cond* does not holds. By the stronger invariant we know there exists such a *rhs* with $(X,\ rhs) \in ES$. Because *Cond* is not true, we know the cardinality of *ES* is *1*. This means *ES* must actually be the set $\{(X,\ rhs)\}$, for which the invariant holds. This allows us to conclude that *Solve X* $(Init\ (UNIV /\!\!/ \approx_A)) = \{(X,\ rhs)\}$ and *invariant* $\{(X,\ rhs)\}$ hold, as needed. □

With this lemma in place we can show that for every equivalence class in $UNIV /\!\!/ \approx_A$ there exists a regular expression.

Lemma 8. *If finite* $(UNIV /\!\!/ \approx_A)$ *and* $X \in UNIV /\!\!/ \approx_A$ *then* $\exists\, r.\ X = \mathcal{L}(r)$*.*

Proof. By the preceding lemma, we know that there exists a *rhs* such that *Solve X* $(Init\ (UNIV /\!\!/ \approx_A))$ returns the equation $X = rhs$, and that the invariant holds for this equation. That means we know $X = \bigcup \mathcal{L}$ ‘ *rhs*. We further know that this is equal to $\bigcup \mathcal{L}$ ‘ $(Arden\ X\ rhs)$ using the properties of the invariant and Lem. 3. Using the validity property for the equation $X = rhs$, we can infer that $rhss\ rhs \subseteq \{X\}$ and because the *Arden* operation removes that X from *rhs*, that $rhss\ (Arden\ X\ rhs) = \varnothing$. This means the right-hand side *Arden X rhs* can only consist of terms of the form $\lambda(r)$. So we can collect those (finitely many) regular expressions *rs* and have $X = \mathcal{L}(+rs)$. With this we can conclude the proof. □

Lem. 8 allows us to finally give a proof for the first direction of the Myhill-Nerode theorem.

Proof (of Thm. 1). By Lem. 8 we know that there exists a regular expression for every equivalence class in $UNIV /\!/ \approx_A$. Since *finals* A is a subset of $UNIV /\!/ \approx_A$, we also know that for every equivalence class in *finals* A there exists a regular expression. Moreover by assumption we know that *finals* A must be finite, and therefore there must be a finite set of regular expressions rs such that $\bigcup \mathit{finals}\ A = \mathcal{L}(\bigplus rs)$. Since the left-hand side is equal to A, we can use $\bigplus rs$ as the regular expression that is needed in the theorem. □

4 Myhill-Nerode, Second Part

We will prove in this section the second part of the Myhill-Nerode theorem. It can be formulated in our setting as follows.

Theorem 2. *Given r is a regular expression, then finite* $(UNIV /\!/ \approx_{\mathcal{L}(r)})$.

The proof will be by induction on the structure of r. It turns out the base cases are straightforward.

Proof (Base Cases). The cases for *NULL*, *EMPTY* and *CHAR* are routine, because we can easily establish that

$$\begin{array}{l} UNIV /\!/ \approx_{\varnothing} = \{UNIV\} \\ UNIV /\!/ \approx_{\{[]\}} \subseteq \{\{[]\}, UNIV - \{[]\}\} \\ UNIV /\!/ \approx_{\{[c]\}} \subseteq \{\{[]\}, \{[c]\}, UNIV - \{[], [c]\}\} \end{array}$$

hold, which shows that $UNIV /\!/ \approx_{\mathcal{L}(r)}$ must be finite. □

Much more interesting, however, are the inductive cases. They seem hard to solve directly. The reader is invited to try.

Our proof will rely on some *tagging-functions* defined over strings. Given the inductive hypothesis, it will be easy to prove that the *range* of these tagging-functions is finite (the range of a function f is defined as $\mathit{range}\ f \stackrel{def}{=} f\ `\ UNIV$). With this we will be able to infer that the tagging-functions, seen as relations, give rise to finitely many equivalence classes of *UNIV*. Finally we will show that the tagging-relations are more refined than $\approx_{\mathcal{L}(r)}$, which implies that $UNIV /\!/ \approx_{\mathcal{L}(r)}$ must also be finite (a relation R_1 is said to *refine* R_2 provided $R_1 \subseteq R_2$). We formally define the notion of a *tagging-relation* as follows.

Definition 3 (Tagging-Relation). *Given a tagging-function tag, then two strings x and y are* tag-related *provided*

$$x =\!tag\!= y \stackrel{def}{=} tag\ x = tag\ y\ .$$

In order to establish finiteness of a set A, we shall use the following powerful principle from Isabelle/HOL's library.

$$\text{If } \mathit{finite}\ (f\ `\ A)\ \text{ and } \mathit{inj\_on}\ f\ A\ \text{ then } \mathit{finite}\ A. \qquad (10)$$

It states that if an image of a set under an injective function f (injective over this set) is finite, then the set A itself must be finite. We can use it to establish the following two lemmas.

Lemma 9. *If finite* $(\mathit{range\ tag})$ *then finite* $(UNIV /\!/ {=}tag{=})$.

Proof. We set in (10), f to be $X \mapsto tag$ ‘ X. We have $range\ f$ to be a subset of Pow $(range\ tag)$, which we know must be finite by assumption. Now f $(UNIV /\!/ {=}tag{=})$ is a subset of $range\ f$, and so also finite. Injectivity amounts to showing that $X = Y$ under the assumptions that X, $Y \in UNIV /\!/ {=}tag{=}$ and $f\,X = f\,Y$. From the assumptions we can obtain $x \in X$ and $y \in Y$ with $tag\ x = tag\ y$. Since x and y are tag-related, this in turn means that the equivalence classes X and Y must be equal. □

Lemma 10. *Given two equivalence relations R_1 and R_2, whereby R_1 refines R_2. If finite $(UNIV /\!/ R_1)$ then finite $(UNIV /\!/ R_2)$.*

Proof. We prove this lemma again using (10). This time we set f to be $X \mapsto \{[\![x]\!]_{R_1} \mid x \in X\}$. It is easy to see that *finite* $(f$ ‘ $UNIV /\!/ R_2)$ because it is a subset of Pow $(UNIV /\!/ R_1)$, which is finite by assumption. What remains to be shown is that f is injective on $UNIV /\!/ R_2$. This is equivalent to showing that two equivalence classes, say X and Y, in $UNIV /\!/ R_2$ are equal, provided $f\,X = f\,Y$. For $X = Y$ to be equal, we have to find two elements $x \in X$ and $y \in Y$ such that they are R_2 related. We know there exists a $x \in X$ with $X = [\![x]\!]_{R_2}$. From the latter fact we can infer that $[\![x]\!]_{R_1} \in f\,X$ and further $[\![x]\!]_{R_1} \in f\,Y$. This means we can obtain a y such that $[\![x]\!]_{R_1} = [\![y]\!]_{R_1}$ holds. Consequently x and y are R_1-related. Since by assumption R_1 refines R_2, they must also be R_2-related, as we need to show. □

Chaining Lem. 9 and 10 together, means in order to show that $UNIV /\!/ \approx_{\mathcal{L}(r)}$ is finite, we have to find a tagging-function whose range can be shown to be finite and whose tagging-relation refines $\approx_{\mathcal{L}(r)}$. Let us attempt the *ALT*-case first.

Proof (ALT-Case). We take as tagging-function

$$tag_{ALT}\ A\ B\ x \overset{def}{=} ([\![x]\!]_{\approx_A}, [\![x]\!]_{\approx_B})$$

where A and B are some arbitrary languages. We can show in general, if *finite* $(UNIV /\!/ \approx_A)$ and *finite* $(UNIV /\!/ \approx_B)$ then *finite* $(UNIV /\!/ \approx_A \times UNIV /\!/ \approx_B)$ holds. The range of $tag_{ALT}\ A\ B$ is a subset of this product set—so finite. It remains to be shown that ${=}tag_{ALT}\ A\ B{=}$ refines $\approx_{A\,\cup\,B}$. This amounts to showing

$$tag_{ALT}\ A\ B\ x = tag_{ALT}\ A\ B\ y \longrightarrow x \approx_{A\,\cup\,B} y$$

which by unfolding the Myhill-Nerode relation is identical to

$$\forall z.\ tag_{ALT}\ A\ B\ x = tag_{ALT}\ A\ B\ y \wedge x\ @\ z \in A \cup B \longrightarrow y\ @\ z \in A \cup B \qquad (11)$$

since both ${=}tag_{ALT}\ A\ B{=}$ and $\approx_{A\,\cup\,B}$ are symmetric. To solve (11) we just have to unfold the definition of the tagging-function and analyse in which set, A or B, the string $x\ @\ z$ is. The definition of the tagging-function will give us in each case the information to infer that $y\ @\ z \in A \cup B$. Finally we can discharge this case by setting A to $\mathcal{L}(r_1)$ and B to $\mathcal{L}(r_2)$. □

The pattern in (11) is repeated for the other two cases. Unfortunately, they are slightly more complicated. In the *SEQ*-case we essentially have to be able to infer that

$$\ldots x\ @\ z \in A \cdot B \longrightarrow y\ @\ z \in A \cdot B$$

using the information given by the appropriate tagging-function. The complication is to find out what the possible splits of x @ z are to be in $A \cdot B$ (this was easy in case of $A \cup B$). To deal with this complication we define the notions of *string prefixes*

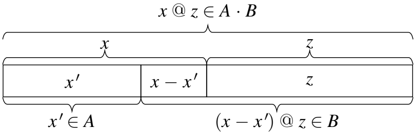
$$x \leq y \overset{def}{=} \exists z.\ y = x \text{ @ } z \qquad\qquad x < y \overset{def}{=} x \leq y \wedge x \neq y$$

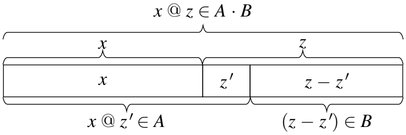
and *string subtraction*:

$$[] - y \overset{def}{=} [] \qquad\qquad x - [] \overset{def}{=} x \qquad\qquad cx - dy \overset{def}{=} \mathit{if}\ c = d\ \mathit{then}\ x - y\ \mathit{else}\ cx$$

where c and d are characters, and x and y are strings.

Now assuming x @ $z \in A \cdot B$ there are only two possible ways of how to 'split' this string to be in $A \cdot B$:





Either there is a prefix of x in A and the rest is in B (first picture), or x and a prefix of z is in A and the rest in B (second picture). In both cases we have to show that y @ $z \in A \cdot B$. For this we use the following tagging-function

$$\mathit{tag}_{SEQ}\ A\ B\ x \overset{def}{=} ([\![x]\!]_{\approx_A}, \{[\![(x - x')]\!]_{\approx_B} \mid x' \leq x \wedge x' \in A\})$$

with the idea that in the first split we have to make sure that $(x - x')$ @ z is in the language B.

Proof (SEQ-Case). If *finite* $(\mathit{UNIV} /\!/ \approx_A)$ and *finite* $(\mathit{UNIV} /\!/ \approx_B)$ then *finite* $(\mathit{UNIV} /\!/ \approx_A \times \mathit{Pow}\ (\mathit{UNIV} /\!/ \approx_B))$ holds. The range of $\mathit{tag}_{SEQ}\ A\ B$ is a subset of this product set, and therefore finite. We have to show injectivity of this tagging-function as

$$\forall z.\ \mathit{tag}_{SEQ}\ A\ B\ x = \mathit{tag}_{SEQ}\ A\ B\ y \wedge x \text{ @ } z \in A \cdot B \longrightarrow y \text{ @ } z \in A \cdot B$$

There are two cases to be considered (see pictures above). First, there exists a x' such that $x' \in A$, $x' \leq x$ and $(x - x')$ @ $z \in B$ hold. We therefore have

$$[\![(x - x')]\!]_{\approx_B} \in \{[\![(x - x')]\!]_{\approx_B} \mid x' \leq x \wedge x' \in A\}$$

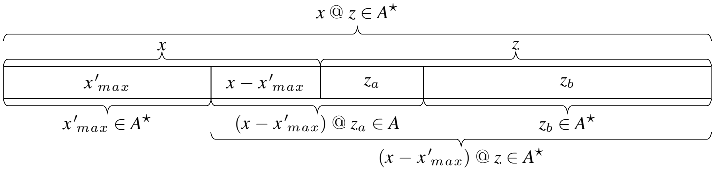
and by the assumption about $\mathit{tag}_{SEQ}\ A\ B$ also

$$[\![(x - x')]\!]_{\approx_B} \in \{[\![(y - y')]\!]_{\approx_B} \mid y' \leq y \wedge y' \in A\}$$

That means there must be a y' such that $y' \in A$ and $[\![(x - x')]\!]_{\approx_B} = [\![(y - y')]\!]_{\approx_B}$. This equality means that $x - x' \approx_B y - y'$ holds. Unfolding the Myhill-Nerode relation and together with the fact that $(x - x')$ @ $z \in B$, we have $(y - y')$ @ $z \in B$. We already know $y' \in A$, therefore y @ $z \in A \cdot B$, as needed in this case.

Second, there exists a z' such that x @ $z' \in A$ and $z - z' \in B$. By the assumption about $\mathit{tag}_{SEQ}\ A\ B$ we have $[\![x]\!]_{\approx_A} = [\![y]\!]_{\approx_A}$ and thus $x \approx_A y$. Which means by the Myhill-Nerode relation that y @ $z' \in A$ holds. Using $z - z' \in B$, we can conclude also in this case with y @ $z \in A \cdot B$. We again can complete the *SEQ*-case by setting A to $\mathcal{L}(r_1)$ and B to $\mathcal{L}(r_2)$. □

The case for *STAR* is similar to *SEQ*, but poses a few extra challenges. When we analyse the case that x @ z is an element in $A^\star$ and x is not the empty string, we have the following picture:



We can find a strict prefix x' of x such that $x' \in A^\star$, $x' < x$ and the rest $(x - x')$ @ $z \in A^\star$. For example the empty string [] would do. There are potentially many such prefixes, but there can only be finitely many of them (the string x is finite). Let us therefore choose the longest one and call it x'_{max}. Now for the rest of the string $(x - x'_{max})$ @ z we know it is in $A^\star$. By definition of $A^\star$, we can separate this string into two parts, say a and b, such that $a \in A$ and $b \in A^\star$. Now a must be strictly longer than $x - x'_{max}$, otherwise x'_{max} is not the longest prefix. That means a 'overlaps' with z, splitting it into two components z_a and z_b. For this we know that $(x - x'_{max})$ @ $z_a \in A$ and $z_b \in A^\star$. To cut a story short, we have divided x @ $z \in A^\star$ such that we have a string a with $a \in A$ that lies just on the 'border' of x and z. This string is $(x - x'_{max})$ @ z_a.

In order to show that x @ $z \in A^\star$ implies y @ $z \in A^\star$, we use the following tagging-function:

$$\mathit{tag}_{STAR}\, A\, x \stackrel{\mathit{def}}{=} \{[\![(x - x')]\!]_{\approx_A} \mid x' < x \wedge x' \in A^\star\}$$

Proof (STAR-Case). If *finite* $(\mathit{UNIV} /\!/ \approx_A)$ then *finite* $(\mathit{Pow}\ (\mathit{UNIV} /\!/ \approx_A))$ holds. The range of $\mathit{tag}_{STAR}\, A$ is a subset of this set, and therefore finite. Again we have to show injectivity of this tagging-function as

$$\forall z.\ \mathit{tag}_{STAR}\, A\, x = \mathit{tag}_{STAR}\, A\, y \wedge x \text{ @ } z \in A^\star \longrightarrow y \text{ @ } z \in A^\star$$

We first need to consider the case that x is the empty string. From the assumption we can infer y is the empty string and clearly have y @ $z \in A^\star$. In case x is not the empty string, we can divide the string x @ z as shown in the picture above. By the tagging-function we have

$$[\![(x - x'_{max})]\!]_{\approx_A} \in \{[\![(x - x')]\!]_{\approx_A} \mid x' < x \wedge x' \in A^\star\}$$

which by assumption is equal to

$$[\![(x - x'_{max})]\!]_{\approx_A} \in \{[\![(y - y')]\!]_{\approx_A} \mid y' < y \wedge y' \in A^\star\}$$

and we know that we have a $y' \in A^\star$ and $y' < y$ and also know $x - x'_{max} \approx_A y - y'$. Unfolding the Myhill-Nerode relation we know $(y - y')$ @ $z_a \in A$. We also know that $z_b \in A^\star$. Therefore y' @ $((y - y')$ @ $z_a)$ @ $z_b \in A^\star$, which means y @ $z \in A^\star$. As the last step we have to set A to $\mathcal{L}(r)$ and complete the proof. □

5 Conclusion and Related Work

In this paper we took the view that a regular language is one where there exists a regular expression that matches all of its strings. Regular expressions can conveniently be defined as a datatype in HOL-based theorem provers. For us it was therefore interesting to find out how far we can push this point of view. We have established in Isabelle/HOL both directions of the Myhill-Nerode theorem.

Theorem 3 (The Myhill-Nerode Theorem)
A language A is regular if and only if finite $(UNIV /\!/ \approx_A)$.

Having formalised this theorem means we pushed our point of view quite far. Using this theorem we can obviously prove when a language is *not* regular—by establishing that it has infinitely many equivalence classes generated by the Myhill-Nerode relation (this is usually the purpose of the pumping lemma [7]). We can also use it to establish the standard textbook results about closure properties of regular languages. Interesting is the case of closure under complement, because it seems difficult to construct a regular expression for the complement language by direct means. However the existence of such a regular expression can be easily proved using the Myhill-Nerode theorem since

$$s_1 \approx_A s_2 \text{ if and only if } s_1 \approx_{\overline{A}} s_2$$

holds for any strings s_1 and s_2. Therefore A and the complement language $\overline{A}$ give rise to the same partitions. Proving the existence of such a regular expression via automata using the standard method would be quite involved. It includes the steps: regular expression ⇒ non-deterministic automaton ⇒ deterministic automaton ⇒ complement automaton ⇒ regular expression.

While regular expressions are convenient in formalisations, they have some limitations. One is that there seems to be no method of calculating a minimal regular expression (for example in terms of length) for a regular language, like there is for automata. On the other hand, efficient regular expression matching, without using automata, poses no problem [10]. For an implementation of a simple regular expression matcher, whose correctness has been formally established, we refer the reader to Owens and Slind [11].

Our formalisation consists of 780 lines of Isabelle/Isar code for the first direction and 460 for the second, plus around 300 lines of standard material about regular languages. While this might be seen as too large to count as a concise proof pearl, this should be seen in the context of the work done by Constable at al [4] who formalised the Myhill-Nerode theorem in Nuprl using automata. They write that their four-member team needed something on the magnitude of 18 months for their formalisation. The estimate for our formalisation is that we needed approximately 3 months and this included the time to find our proof arguments. Unlike Constable et al, who were able to follow the proofs from [6], we had to find our own arguments. So for us the formalisation was not the bottleneck. It is hard to gauge the size of a formalisation in Nurpl, but from what is shown in the Nuprl Math Library about their development it seems substantially larger than ours. The code of ours can be found in the Mercurial Repository at http://www4.in.tum.de/~urbanc/regexp.html.

Our proof of the first direction is very much inspired by *Brzozowski's algebraic method* used to convert a finite automaton to a regular expression [3]. The close connection can be seen by considering the equivalence classes as the states of the minimal automaton for the regular language. However there are some subtle differences. Since we identify equivalence classes with the states of the automaton, then the most natural choice is to characterise each state with the set of strings starting from the initial state leading up to that state. Usually, however, the states are characterised as the strings starting from that state leading to the terminal states. The first choice has consequences about how the initial equational system is set up. We have the λ-term on our 'initial state', while Brzozowski has it on the terminal states. This means we also need to reverse the direction of Arden's Lemma.

We briefly considered using the method Brzozowski presented in the Appendix of [3] in order to prove the second direction of the Myhill-Nerode theorem. There he calculates the derivatives for regular expressions and shows that for every language there can be only finitely many of them (if regarded equal modulo ACI). We could have used as tagging-function the set of derivatives of a regular expression with respect to a language. Using the fact that two strings are Myhill-Nerode related whenever their derivative is the same, together with the fact that there are only finitely such derivatives would give us a similar argument as ours. However it seems not so easy to calculate the set of derivatives modulo ACI. Therefore we preferred our direct method of using tagging-functions. This is also where our method shines, because we can completely side-step the standard argument [7] where automata need to be composed, which as stated in the Introduction is not so easy to formalise in a HOL-based theorem prover. However, it is also the direction where we had to spend most of the 'conceptual' time, as our proof-argument based on tagging-functions is new for establishing the Myhill-Nerode theorem. All standard proofs of this direction use arguments over automata.

References

1. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 24–40. Springer, Heidelberg (2002)
2. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 147–163. Springer, Heidelberg (2009)
3. Brzozowski, J.A.: Derivatives of Regular Expressions. J. ACM 11, 481–494 (1964)
4. Constable, R.L., Jackson, P.B., Naumov, P., Uribe, J.C.: Constructively Formalizing Automata Theory. In: Proof, Language, and Interaction, pp. 213–238. MIT Press, Cambridge (2000)
5. Filliâtre, J.-C.: Finite Automata Theory in Coq: A Constructive Proof of Kleene's Theorem. Research Report 97–04, LIP - ENS Lyon (1997)
6. Hopcroft, J.E., Ullman, J.D.: Formal Languages and Their Relation to Automata. Addison-Wesley, Reading (1969)
7. Kozen, D.: Automata and Computability. Springer, Heidelberg (1997)
8. Kraus, A., Nipkow, T.: Proof Pearl: Regular Expression Equivalence and Relation Algebra. To appear in Journal of Automated Reasoning (2011)
9. Nipkow, T.: Verified lexical analysis. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 1–15. Springer, Heidelberg (1998)
10. Owens, S., Reppy, J., Turon, A.: Regular-Expression Derivatives Re-Examined. Journal of Functional Programming 19(2), 173–190 (2009)
11. Owens, S., Slind, K.: Adapting Functional Programs to Higher Order Logic. Higher-Order and Symbolic Computation 21(4), 377–409 (2008)

LCF-Style Bit-Blasting in HOL4

Anthony C.J. Fox

Computer Laboratory, University of Cambridge, Cambridge, UK

Abstract. This paper describes a new proof tool for deciding bit-vector problems in HOL4. The approach is based on "bit-blasting", wherein word expressions are mapped into propositional formulas, which are then handed to a SAT solver. Significantly, the implementation uses the LCF approach, which means that the soundness of the tool is guaranteed by the soundness of HOL4's logical kernel.

1 Introduction

Interactive theorem provers are often used in areas such as hardware design, computer architectures, compilers, operating systems, protocols and cryptography. Inevitably these domains provide an abundant source of bit-vector problems, and users would like the ability to prove these goals automatically. For example, consider the formula

$$(a_{(:32)} \;\&\&\; \mathbf{3} = \mathbf{0}) \Rightarrow ((a + \mathbf{4}) \;\&\&\; \mathbf{3} = \mathbf{0})$$

where $a_{(:32)}$ indicates that a is a 32-bit word and && is bitwise-and. This theorem shows that adding four to a word aligned memory address does not break the alignment property. Such goals are potentially challenging for HOL4 users, but this goal can now be solved fully automatically and quickly (0.05 s).

The tool presented here uses an established technique called *bit-blasting*. Although the implementation is much simpler than highly advanced bit-vector decision procedures (such as [3]), the tool is implemented in an LCF style, which is of great advantage with respect to ensuring logical soundness. The principle design objective was to produce a simple tool that can handle many "small but somewhat tricky" bit-vectors problems that often arise during interactive proofs. In this sense the tool has already been very successful. Recently it has been used to great effect by Magnus Myreen during machine code verification as part of the Jitawa project, see [5].

There will be bit-vector problems that are too complex for the tool to handle quickly. Nevertheless, complex problems can often be tackled with some human guidance.[1] As with provers such as PVS and Isabelle, HOL4 users can also call external high-performance proof tools, treating these tools as *oracles*. Recently Tjark Weber has integrated SMT solvers with bit-vector capabilities into HOL, see [2]. Theorems that are tagged as coming from oracles are considered undesirable in HOL, since they do not offer the high assurance of LCF-style proofs. Weber uses our new LCF procedure to safely reconstruct SMT bit-vector proofs.

[1] Users can discover and apply helpful abstractions, case-splits and simplifications.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 357–362, 2011.

2 Representation of Bit-Vectors

Bit-vectors can be represented as *finite Cartesian products* $\mathbb{B}^n$ where n is the finite, fixed *width* (or length) of the bit-vector. At first glance, it does not seem possible to *directly* represent the set $\mathbb{B}^n$ using HOL4's simple type system. However, in [4] John Harrison showed that parametric polymorphism can be used to specify vector widths. Using Harrison's approach bit-vectors are represented by the type $\mathsf{bool}[\alpha]$ and the word length is given by the term $\dim(:\alpha)$, where α is a type variable. Readers are referred to Harrison's paper for the details of this approach.

Note that HOL4's parser and pretty-printer support *numeric types*, which makes it easy to work with concrete bit-vector instances. For example, 32-bit words are represented by $\mathsf{bool}[32]$ and we have $\dim(:32) = 32$.

This representation of bit-vectors has been used in HOL4 since 2005 and has worked well in practice. In particular we gain the benefits of exploiting HOL4's well established support for parametric polymorphism. There are some limitations — the type system is not intelligent with regard to result type of word extraction and concatenation, but some extra parsing tool support is provided to help address this drawback.

3 Bit-Vector Operations

Having defined a representing type for bit-vectors, one can define a collection of standard operations. These generally split into three camps:

1. *Arithmetic operations*: $+$, unary $-$, binary $-$, $\times$, $\div$, mod, $<$, $\leq$, $>$, and $\geq$. In some cases there are signed (2's complement) and unsigned variants.
2. *Bitwise/logical operations*: $\neg$ (bitwise NOT), && (bitwise AND), $\|$ (bitwise OR), $\oplus$ (bitwise XOR), shifts and rotations.
3. *Casting maps*: unsigned maps to and from $\mathbb{N}$ (w2n and n2w), signed maps to and from $\mathbb{Z}$ (w2i and i2w), unsigned and signed word-to-word maps (w2w and sw2sw), and word extraction and concatenation.

These and other operations are defined through the use of a finite Cartesian product *binder* FCP:

$$(\,\mathrm{FCP}\,) : (\mathsf{num} \to \beta) \to \beta[\alpha]$$

and a *projection* function

$$(\,'\,) : \beta[\alpha] \to \mathsf{num} \to \beta\ .$$

For bit-vectors, β is specialised to bool. The FCP binder constructs a Cartesian product from a function.[2] The bitwise operators are easy to define; for example, bitwise conjunction && : $\mathsf{bool}[\alpha] \to \mathsf{bool}[\alpha] \to \mathsf{bool}[\alpha]$ is defined as follows:

$$a \mathbin{\&\&} b =_{def} \mathrm{FCP}\ i.\ (a\ '\ i) \wedge (b\ '\ i)\ .$$

[2] In HOL4, the binder syntax FCP $i.\ f\ i$ is used to denote FCP $(\lambda i.\ f\ i)$.

It is also easy to prove that

$$\forall a, b : \mathsf{bool}[\alpha]\ i : \mathsf{num}.\ \ i < \dim(:\alpha) \Rightarrow ((a \mathbin{\&\&} b)\ '\ i = (a\ '\ i) \wedge (b\ '\ i))\ .$$

The arithmetic operations are defined using the maps w2n and n2w. For example, addition is defined as follows:

$$a + b =_{def} \mathrm{n2w}(\mathrm{w2n}(a) + \mathrm{w2n}(b))\ .$$

Here bit-vector addition is on the left-hand side and natural number addition is on the right-hand side. The natural number mappings are defined as follows:

$$\mathrm{w2n}(a : \mathsf{bool}[\alpha]) =_{def} \sum_{0 \le i < \dim(:\alpha)} \textbf{if}\ a\ '\ i\ \textbf{then}\ 2^i\ \textbf{else}\ 0$$

and

$$\mathrm{n2w}(n) =_{def} \mathrm{FCP}\ i.\ (n \operatorname{div} 2^i) \bmod 2 = 1\ .$$

To facilitate bit-blasting, the following theorems are proved:

$$\begin{aligned} &\vdash \forall x\, y.\ x + y = \mathrm{FCP}\ i.\ \ \mathrm{Sum}\ \ i\ (\lambda i.\ x\ '\ i)\ (\lambda i.\ y\ '\ i)\ \bot \\ &\vdash \forall x\, y.\ x - y = \mathrm{FCP}\ i.\ \ \mathrm{Sum}\ \ i\ (\lambda i.\ x\ '\ i)\ (\lambda i.\ \neg(y\ '\ i))\ \top \\ \vdash \forall x\, y.\ x_{(:\alpha)} < y &= \neg(\mathrm{Carry}\ (\dim(:\alpha))\ (\lambda i.\ x\ '\ i)\ (\lambda i.\ \neg(y\ '\ i))\ \top) \end{aligned}$$

where $\top$ represents true, $\bot$ is false, $<$ is *unsigned* less-than (<+ in HOL4) and

$$\mathrm{Carry}, \mathrm{Sum} : \mathsf{num} \to (\mathsf{num} \to \mathsf{bool}) \to (\mathsf{num} \to \mathsf{bool}) \to \mathsf{bool} \to \mathsf{bool}$$

are primitive recursive, circuit-like specifications for a ripple-carry adder. At present bit-blasting of division is not supported and *general* multiplication has been constrained to small word sizes (at the moment less than nine bits). However, multiplication by a constant is supported at all word lengths, e.g. $\mathbf{3} \cdot x$.

4 Bit-Blasting

Many common bit-vector problems can be readily solved using simplification, where collections of algebraic equations and rules are applied as rewrites. This approach works especially well when working with bit-vector operations that come exclusively from one category, i.e. all arithmetic or all bitwise. For example, the following are *automatically* proved via standard word simplification:

$$\begin{gathered} b \cdot \mathbf{2} + \mathbf{4} \cdot a - b + a - b = \mathbf{5} \cdot a, \\ a \oplus (b \parallel a \parallel \neg b) = (\neg(a \mathbin{\&\&} \neg a) \parallel a \parallel b) \mathbin{\&\&} \neg a\ . \end{gathered}$$

Things get harder when operations are freely mixed, for example:

$$\begin{gathered} ((a_{(:32)} + \mathrm{w2w}(b_{(:8)}))[7:0] = a[7:0] + b, \\ \neg(x\ '\ 0) \Rightarrow ((\mathbf{17} \cdot x_{(:8)}) \mathbin{\&\&} \mathbf{6} = \mathbf{7} \cdot x) \end{gathered}$$

Here, w2w : $\mathsf{bool}[\alpha] \to \mathsf{bool}[\beta]$ is a word-to-word mapping (this zero extends on expansion and truncates on contraction) and $x[i:j]$ represents extraction over a bit range. These are the sorts of goals that users may find hard to prove manually but that bit-blasting can handle with ease.

Implementation. As with many proof procedures in HOL4, the underlying LCF implementation is in the form of a *conversion*, which converts a term into an equality theorem, where the original term occurs on the left-hand side. A decision procedure succeeds if the right-hand side after conversion is $\top$.

To demonstrate the implementation, the following example formula is used:

$$(a_{(:2)} < b) \wedge (b < c) \wedge (c < d) \Rightarrow (a \parallel (\mathbf{3} \cdot d) = \mathbf{1}) \ .$$

The domain is 2-bit words (i.e. $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$). The chain of orderings in the antecedent imply that a must be the infimum $\mathbf{0}$ and d must be the supremum $\mathbf{3}$ and hence the equality is known to hold (since $\mathbf{0} \parallel (\mathbf{3} \cdot \mathbf{3}) = \mathbf{1}$).

The decision procedure calls the conversion `BBLAST_CONV`, which starts by applying standard bit vector simplifications.[3] The next stage is identifying sub-terms that are amenable to bit-blasting — here there are four such sub-terms:

$$a < b \qquad b < c \qquad c < d \qquad a \parallel (\mathbf{3} \cdot d) = \mathbf{1}$$

The inequalities are expanded as follows:

$$\begin{aligned} a_{(:2)} < b &\mapsto \neg(\text{Carry } 2\ (\lambda i.\ a\ '\ i)\ (\lambda i.\ \neg(b\ '\ i))\ \top) \\ &\mapsto \neg(a\ '\ 1 \wedge \neg b\ '\ 1 \vee (a\ '\ 1 \vee \neg b\ '\ 1) \wedge (a\ '\ 0 \vee \neg b\ '\ 0)) \ . \end{aligned}$$

The equality (consequent) sub-term is more complicated. The multiplication by a constant is eliminated using the conversion $\mathbf{3} \cdot x \mapsto x \ll 1 + x$ where $\ll$ is logical shift-left. The term is then rewritten, introducing the FCP binder wherever possible. For example, we use the definition of shift-left:

$$x_{(:\alpha)} \ll n =_{def} \text{FCP } i.\ i < \dim(:\alpha) \wedge n \leq i \wedge (x\ '\ (i - n)) \ .$$

The result at this stage is of the form:

$$(a \parallel (\mathbf{3} \cdot d) = \mathbf{1}) \mapsto (\text{FCP } i.\ a\ '\ i \vee (\text{FCP } i.\ \text{Sum } i\ \ldots)\ '\ i) = (\text{FCP } i.\ i = 0) \ .$$

In general, word equalities will be converted into the form FCP f = FCP g for some pair of functions $f, g : \mathsf{num} \to \mathsf{bool}$. The next stage is to use the theorem:

$$(\text{FCP } f)_{(:2)} = \text{FCP } g \Leftrightarrow (f(0) = g(0)) \wedge (f(1) = g(1)) \ .$$

We now try to symbolically evaluate f and g at each bit position. Care is taken to perform this evaluation efficiently – there is a preliminary stage that iteratively generates sets of rewrites for Sum and Carry sub-terms.[4] With our example, the evaluation at position zero gives us: $f(0) = a\ '\ 0 \vee d\ '\ 0$ and $g(0) = \top$. If $f(0) = g(0)$ rewrites to false then we can quit early, knowing that the entire word equality is false, but with this example we move on to the next bit position. After some basic Boolean simplification, we have the conversion:

$$(a \parallel (\mathbf{3} \cdot d) = \mathbf{1}) \mapsto (a\ '\ 0 \vee d\ '\ 0) \wedge \neg(a\ '\ 1 \vee (d\ '\ 0 \Leftrightarrow \neg d\ '\ 1)) \ .$$

[3] Simplification does not alter our example goal. However, at best simplification will convert terms to $\top$ or $\bot$, in which case bit-blasting will not occur.

[4] Note that Sum terms can be nested, so the order of rewrite generation is important.

A SAT solver is called on this proposition, which reveals that it is *contingent*, i.e. we cannot rewrite it to $\top$ or $\bot$. Having converted all of the original sub-terms, we are left with the proposition:

$$\begin{gathered}\neg(a\ '\ 1 \wedge \neg b\ '\ 1 \vee (a\ '\ 1 \vee \neg b\ '\ 1) \wedge (a\ '\ 0 \vee \neg b\ '\ 0)) \wedge \\ \neg(b\ '\ 1 \wedge \neg c\ '\ 1 \vee (b\ '\ 1 \vee \neg c\ '\ 1) \wedge (b\ '\ 0 \vee \neg c\ '\ 0)) \wedge \\ \neg(c\ '\ 1 \wedge \neg d\ '\ 1 \vee (c\ '\ 1 \vee \neg d\ '\ 1) \wedge (c\ '\ 0 \vee \neg d\ '\ 0)) \Rightarrow \\ (a\ '\ 0 \vee d\ '\ 0) \wedge \neg(a\ '\ 1 \vee (d\ '\ 0 \Leftrightarrow \neg d\ '\ 1))\ .\end{gathered}$$

This time calling the SAT solver gives us $\top$ and the decision procedure succeeds.

A very useful facility of this SAT based decision procedure is the ability to print counterexamples. For example, calling the procedure on $a \leq a + b_{(:4)}$ gives the counterexample $a \mapsto \mathbf{12}$ and $b \mapsto \mathbf{4}$.

Performance. Unsurprisingly, bit-blasting decision procedures can encounter severe complexity problems. Figure 1 illustrates a worst-case scenario, showing the time required to apply BBLAST_CONV to the following sequence of terms:

$$x = a + b, \quad x = a + b + c, \quad x = a + b + c + d, \quad x = a + b + c + d + e\ .$$

The second term took just over four minutes to convert for 128-bit words using a 2.5 GHz Core 2 Duo machine with 4 GB of RAM. The resulting theorem is very large and calling the SAT solver (to no avail) took up 82 s.[5] However, this gives a false impression. In practice, run-times are typically much more respectable. Consider, the following equations:

$$(\mathbf{193} \cdot a)\ \&\&\ \mathbf{7} = \mathbf{7}\ \&\&\ a \tag{1}$$

$$(a\ \&\&\ b) + (a \parallel b) = a + b \tag{2}$$

$$(\mathbf{8} \cdot a + (b\ \&\&\ \mathbf{7})) \ggg 3 = a\ \&\&\ (-\mathbf{1} \ggg \mathbf{3}) \tag{3}$$

where $\ggg$ is logical shift-right. Figure 2 shows the timings for these problems. Equation 1 has two nested additions (from multiplying by **193**) but this time the 128-bit case takes around 45 s to solve. The difference here is that we can rewrite to $\top$ at each bit position, and this avoids passing on a large term to the SAT solver. In Equation 2 we have two additions but this time they are not nested, so the goal is less complex. In Equation 3 we have one addition, with that addition becoming "simple" after bit position three (the bit value of the second argument becomes false). Equation 3, and goals without additions, scale very well to large word sizes. When working with typical *machine* word sizes (i.e. 32-bit and 64-bit words), the run-times for all three problems are in the order of seconds.

5 Summary

This paper has demonstrated that it is possible to implement a useful, practical and efficient LCF-style proof procedure for bit-vectors in HOL4. The source code

[5] The timings do not include the printing of terms. For obvious reasons the maximum print-depth must be limited when working with very large terms.

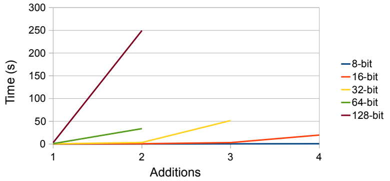


Fig. 1. Nested additions (conversion)

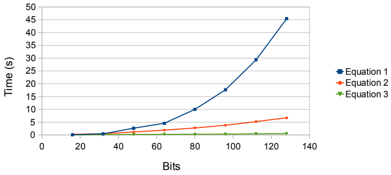


Fig. 2. Typical problems

is distributed with HOL4. This development has been made possible through the work of Hasan Amjad and Tjark Weber in integrating modern SAT solvers (zChaff and MiniSat) into HOL provers using the LCF approach, see [1]. Michael Norrish's DPLL based proof procedure (described in the HOL4 Tutorial, see `hol.sf.net`) is also used to quickly handle small propositions.

There are circumstances when bit-blasting is not appropriate and users will find that traditional methods (simplification and lemma construction) are required — either for efficiency reasons or when proving general results, i.e. for arbitrary word sizes. The proof technique relies on formulating and symbolically evaluating a bit-stream function $f : \mathsf{num} \rightarrow \mathsf{bool}$. It is this requirement that ultimately determines the scope of the procedure. For example, goals involving $x[m : n]$, $\mathrm{w2n}(w)$, $\mathrm{w2i}(w)$, $\mathrm{n2w}(n)$ and $\mathrm{i2w}(i)$, where m, n and i are not constant values, will require reasoning over $\mathbb{N}$ or $\mathbb{Z}$. Nevertheless, the tool does have excellent coverage and it automatically handles many common goals that users could easily waste many hours solving manually. The tool can even handle basic existential goals, e.g. it proves $\exists\, x\, y.\ (x \cdot y = \mathbf{12}_{(:8)}) \wedge x < y$ in 2.3 s. One possible area for performance improvements is to automatically introducing abstractions, e.g. replacing "costly but non-critical" sub-expressions with variables.

Many thanks to Mike Gordon, Magnus Myreen and Tjark Weber for helpful feedback and assistance.

References

1. Amjad, H., Weber, T.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2007)
2. 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)
3. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
4. Harrison, J.V.: A HOL Theory of Euclidean Space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
5. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: Geuvers, H., et al. (eds.) ITP 2011. LNCS, vol. 6898, pp. 357–362. Springer, Heidelberg (2011)

Lem: A Lightweight Tool for Heavyweight Semantics

Scott Owens[1], Peter Böhm[1], Francesco Zappa Nardelli[2], and Peter Sewell[1]

[1] University of Cambridge
[2] INRIA
www.cl.cam.ac.uk/users/so294/lem

Abstract. Many ITP developments exist in the context of a single prover, and are dominated by proof effort. In contrast, when applying rigorous semantic techniques to realistic computer systems, engineering the definitions becomes a major activity in its own right. Proof is then only one task among many: testing, simulation, communication, community review, etc. Moreover, the effort invested in establishing such definitions should be re-usable and, where possible, irrespective of the local proof-assistant culture. For example, in recent work on processor and programming language concurrency (x86, Power, ARM, C++0x, CompCertTSO), we have used Coq, HOL4, Isabelle/HOL, and Ott—often using multiple provers simultaneously, to exploit existing definitions or local expertise.

In this paper we describe Lem, a prototype system specifically designed to support pragmatic engineering of such definitions. It has a carefully designed source language, of a familiar higher-order logic with datatype definitions, inductively defined relations, and so on. This is typechecked and translated to a variety of programming languages and proof assistants, preserving the original source structure (layout, comments, etc.) so that the result is readable and usable. We have already found this invaluable in our work on Power, ARM and C++0x concurrency.

1 Motivation

Mechanised proof assistants such as ACL2 [1], Coq [6], HOL4 [9], HOL Light [8], Isabelle/HOL [10], PVS [12], and Twelf [19] are becoming important tools for Computer Science. In many applications of these tools, the majority of effort is devoted to proof, and that is rightly a main focus of their developers. This focus leads each of these systems to have its own logic, various mechanisms for making mathematical definitions, and extensive support for machine-checked interactive and/or automated reasoning.

In some applications, however, the *definitions* themselves, of types, functions, and relations, are a major focus of the work. This is often the case when modelling key computational infrastructure: network protocols, programming languages, multiprocessors, and so on. For example, we have worked on TCP [4,13], Optical Networking [3], Java Module Systems [18], the semantics of an OCaml fragment [11], concurrency for C and C++ [2,5,16], and the semantics of x86,

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 363–369, 2011.

POWER and ARM multiprocessors [14,15]; and there are numerous examples by other groups (far too many to cite here). In each of these cases, considerable effort was required to establish the definitions of syntax and semantics, including analysis of informal specifications, empirical testing, and proof of metatheory. These definitions can be large: for example, our TCP specification is around 10 000 non-comment lines of HOL4. At this scale, the activity of working with the definitions becomes more like developing software than defining small calculi: one has to refactor, test, coordinate between multiple people, and so on, and all of this should, as far as possible, be complete before one embarks on any proof.

Moreover, in such work a proof assistant is just one piece of a complex project, involving production typesetting, testing infrastructure, code generation, and tools for embedding source-language terms into the prover. Sometimes there is no proof activity, but great benefits arise simply from working in typechecked and typeset mathematics; sometimes there is mechanised symbolic evaluation or code generation for testing and prototyping; sometimes there is hand proof or a mixture of hand and mechanised proof; and sometimes there is the classic full mechanised proof supported by provers.

Ideally, the results of such work should be made widely available in a *reusable* form, so that other groups can build on them and so that the field can eventually converge on standard models for the relatively stable aspects of the computational environment in which we work. Unfortunately, at present such re-use is highly restricted for two reasons. Firstly, the field is partitioned into schools around each prover: the difficulty in becoming fluent in their use means that very few people can use more than one tool effectively. Indeed, even within some of our own projects we have had to use several provers due to differing local expertise. This variation makes it hard to compare the results of even carefully specified benchmarks, such as the POPLmark challenge.

Secondly, the differences between the provers mean that it is a major and error-prone task to port a development—or even just its definitions—from one system to another. In some cases this is for fundamental reasons: definitions which make essential use of the dependent types of Coq may be hard or impossible to practically port to HOL4. However, many of the examples cited above are logically undemanding: they have no need for dependent types, the differences between classical and constructive reasoning are not particularly relevant, and there is often little or no object-language variable binding (of course this does not apply for formalisation of rich type theories). They do make heavy use of basic discrete mathematics and "programming language" features: sets and set comprehensions; first-order logic; and inductive types and records with functions and relations over them. Thus, the challenge is one of robustly translating between the concrete syntax and definition styles of the different proof assistants.

2 Portable Definitions with Lem

We have designed a language, Lem, for writing, managing, and publishing large scale semantic definitions, for use as an intermediate language when generating definitions from domain-specific tools, and for use as an intermediate language for porting definitions between existing provers. Our implementation can currently typecheck Lem sources, and generate HOL4, Isabelle/HOL, OCaml, and LaTeX (the latter drawing on Wansborough's HOLDoc tool design). Development of a Coq backend is in progress. We are already using Lem in our research: we developed a semantics for multiprocessor concurrency on the POWER architecture [14] in Lem, and our semantics for C++0x [2,5] concurrency has been ported from Isabelle/HOL to Lem.

Semantically, we have designed Lem to be roughly the intersection of common functional programming languages and higher-order logics, as we regard this as a sweet spot: expressive enough for the applications we mention above, yet familiar and relatively easy to translate into the various provers; there is intentionally no logical novelty here. Lem has a simple type theory with primitive support for recursive and higher-order functions, inductive relations, n-ary tuples, algebraic datatypes, record types, type inference, and top-level polymorphism. It also includes a type class mechanism broadly similar to Isabelle's and Haskell's (without constructor classes). It differs from the internal logics of HOL4 or Isabelle/HOL principally in having type, function and relation definitions as part of the language rather than encoded into it: the Lem type system is formally defined (using Ott [17]) in terms of the user-level syntax.

The novelty is rather in the detailed design and implementation, to ensure the following four important pragmatic properties. We can achieve all of these goals more easily than one could in context of a prover implementation because we are not constrained to use an intermediate representation suitable for the implementation of a proof kernel (e.g., explicitly typed lambda terms), and because we are building a lightweight stand-alone tool, without a large legacy codebase.

1. Readability of source files. Lem syntactically resembles OCaml and F#, giving us a popular and readable syntax. It includes nested modules (but not functors), recursive type and function definitions, record types, type abbreviations, and pattern matching. It has additional syntax for quantifiers, including restricted quantifiers ($\forall x \in S.\ Px$), set comprehension, and inductive relations. For example, here is an extract from our POWER model:

```
let write_reaching_coherence_point_action m s w =
  let writes_past_coherence_point' =
    s.writes_past_coherence_point union {w} in
  (* make write before other writes to this address not past coherence *)
  let coherence' = s.coherence union
      { (w,wother) | forall (wother IN (writes_not_past_coherence s))
      | (not (wother = w)) && (wother.w_addr = w.w_addr)}  in
  <| s with coherence = coherence';
            writes_past_coherence_point = writes_past_coherence_point' |>
let sem_of_instruction i ist =
  match i with
  | Padd  set rD rA rB  -> op3regs   Add set rD rA rB ist
  | Psub  set rD rA rB  -> op3regs   Sub set rD rB rA ist (* swap args *)
  end
```

We do not always follow OCaml: for example, Lem uses curried data constructors instead of tupled ones, and it uses `<|` and `|>` for records, saving `{` and `}` for set comprehensions. Type classes provide principled support for overloading.

Lem does not at present include support for arbitrary user-defined syntax, as provided by Ott [17] and (to a greater or lesser extent) by several proof assistants. Lem and Ott have complementary strengths: Ott is particularly useful for defining semantics as inductively defined relations over a rich user syntax, but has limited support for logic, sets, and function definitions, whereas Lem is the converse. We envisage refactoring the Ott implementation, which currently generates Coq, HOL, and Isabelle/HOL code separately, to instead generate Lem code and leave the prover-specific output to the Lem tool. In the longer term, a metalanguage that combines both is highly desirable.

2. Taking the source text seriously. *Explaining* the definitions is a key aspect of the kind of work we mention above. We need to produce production-quality typesetting, of the complete definitions in logical order and of various excerpts, in papers, longer documents, and presentations. As all these have to be maintained as the definitions evolve, the process must be automated, without relying on cut-and-paste or hand-editing of generated LaTeX code. Moreover, it is essential to give the user control of layout. Here again the issues of large-scale definitions force our design: in some cases, especially for small definitions, pretty printing from a prover internal representation can do a good enough job, but manual formatting choices were necessary to make (e.g.) our C++0x memory model readable. Accordingly, we preserve all source-file formatting, including line breaks, indentation, comments, and parentheses, in the generated code. This lets us generate corresponding LaTeX code, e.g. for the previous example:

```
let write_reaching_coherence_point_action m s w =
  let writes_past_coherence_point′ =
    s.writes_past_coherence_point ∪ {w} in
  (* make write before other writes to this address not past coherence *)
  let coherence′ = s.coherence ∪
      {(w, wother)|∀wother∈(writes_not_past_coherence s)
      | (¬ (wother = w)) ∧ (wother.w_addr = w.w_addr)} in
  ⦇s with coherence = coherence′;
          writes_past_coherence_point = writes_past_coherence_point′⦈

let sem_of_instruction i ist =
  match i with
  | PADD set rD rA rB → op3regs ADD set rD rA rB ist
  | PSUB set rD rA rB → op3regs SUB set rD rB rA ist (* swap args *)
  end
```

It also ensures that the generated prover and OCaml code is human-readable in its own right.

3. Support for execution. *Exploring* such definitions, and *testing* conformance between specifications and deployed implementations (and between specifications at different levels of abstraction), is also a central aspect of our work; both need some way to make the definitions executable. In previous work with various colleagues we have built hand-crafted symbolic evaluators within HOL4 [3,4,13,15], interpreters from code extracted from Coq [16], and memory model exploration tools from code generated from Isabelle/HOL [2]. Lem supports several constructs which cannot in general be executed, e.g., quantification in propositions and set comprehensions, but Lem can generate OCaml code where the range is restricted to a finite set (otherwise OCaml generation fails). This has been invaluable for our POWER memory model exploration tool [14].

4. Quick parsing and type checking with good error messages. This is primarily a matter of careful engineering, using conventional programming-language techniques. Lem is a batch-mode tool in the style of standard compilers, rather than focussed on interactive use, in the typical proof-assistant style.

3 Implementation

Our Lem implementation is written in OCaml, using Ott to specify the concrete syntax, and it loosely follows the architecture of a traditional compiler. The central data structure is a typed abstract syntax tree (AST), and processing follows 4 phases: (1) source files are lexed and parsed into untyped ASTs; (2) the untyped ASTs are type checked and converted into typed ASTs; (3) typed-AST-to-typed-AST transformations remove language features that are not present in the target (e.g., the removal of type classes by introducing dictionary passing for OCaml and HOL4); and (4) the transformed, typed AST is printed in the target language syntax. We try to make the printing step as simple as possible, and uniform across the various back-ends, by handling all of the complexities of translation in (3). The untyped and typed ASTs contain all of the whitespace (both indentation and line breaks) and comments of the original source file; the step (4) printer uses these instead of a pretty printing algorithm for layout.

The logical design of Lem makes the basic translation to a variety of targets straightforward. The standard libraries of our various targets have differing data representations and interfaces. For each desired feature (e.g., finite maps, or bit vectors), we design an interface for Lem, and specify how that interface is to be translated for each target. This is similar to Ott's *hom* functionality; however, here we typecheck the translation specifications to ensure that the generated code is well-formed.

4 Future work

We are actively developing Lem: our immediate goal is to finish and polish the existing backends (including the in-progress Coq backend). Also of interest is a HOL4-to-Lem translation (allowing us to automatically port, for example, Fox's detailed ARM instruction semantics [7] to other provers) and we would like Coq-to-Lem and Isabelle/HOL-to-Lem translations, which will need expertise in the front-end implementations of those systems. Lem does not currently support OCaml generation for inductively defined relations (although one can sometimes use the Isabelle backend and then apply its code generation mechanism). Ultimately, we would like to directly generate OCaml that searches for derivations;

this will be particularly useful in conjunction with Ott, for running test and example programs directly on an operational semantics.

Although Lem is primarily a design and engineering project, it would benefit from a rigorous understanding of exactly how the semantics of the source and target logics relate to each other, for the fragments we consider. In particular, when multiple provers are used to verify properties of a Lem-specified system, we would like a semantic justification that the resulting definitions have the same meaning, and that a lemma verified in one prover can be used in another. There have been several projects that port low-level proofs between provers (a very different problem to the readable-source-file porting that we consider here); while this approach yields the right guarantees, we expect it would be very challenging because the various backends can transform the same definition differently (e.g., keeping type classes for Isabelle, but not for HOL4).

References

1. ACL2 Version 4.2 (2011), `http://www.cs.utexas.edu/~moore/acl2/`
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011, pp. 55–66. ACM, New York (2011)
3. Biltcliffe, A., Dales, M., Jansen, S., Ridge, T., Sewell, P.: Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In: ICNP 2006, pp. 117–126. IEEE, Los Alamitos (2006)
4. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In: SIGCOMM 2005, pp. 265–276. ACM, New York (2005)
5. Blanchette, J.C., Weber, T., Batty, M., Owens, S., Sarkar, S.: Nitpicking C++ concurrency. In: PPDP 2011. ACM, New York (to appear, 2011)
6. The Coq proof assistant, v.8.3 (2011), `http://coq.inria.fr/`
7. 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)
8. Harrison, J.: HOL Light (2011), `http://www.cl.cam.ac.uk/~jrh13/hol-light/`
9. The HOL 4 system, Kananaskis-6 release (2011), `http://hol.sourceforge.net/`
10. Isabelle 2011 (2011), `http://isabelle.in.tum.de/`
11. Owens, S.: A sound semantics for OCaml$_{light}$. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008)
12. PVS 5.0 (2011), `http://pvs.csl.sri.com/`
13. Ridge, T., Norrish, M., Sewell, P.: A rigorous approach to networking: TCP, from implementation to protocol to service. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 294–309. Springer, Heidelberg (2008)
14. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI 2011. ACM, New York (to appear, 2011)
15. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86 multiprocessor machine code. In: POPL 2009, pp. 379–391. ACM, New York (2009)

16. Šečík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL 2011, pp. 43–54. ACM, New York (2011)
17. Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. JFP 20(1) (January 2010)
18. Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: core design and semantic definition. In: OOPSLA 2007, pp. 499–514. ACM, New York (2007)
19. Twelf 1.5 (2011), `http://twelf.plparty.org/wiki/Main_Page`

Composable Discovery Engines for Interactive Theorem Proving

Phil Scott and Jacques Fleuriot

Centre for Intelligent Systems and their Applications, Informatics Forum,
University of Edinburgh, 10 Crichton Street, Edinburgh, UK, EH8 9AB
phil.scott@ed.ac.uk, jdf@inf.ed.ac.uk

Abstract. We describe a framework to integrate discovery engines with interactive theorem proving, and define an algebra for composing them. Discovery engines can be tailored to specific domains and invoked concurrently as the user writes the proof. The engines collaborate with the user by inferring facts from the current goal context, and providing new theorems to advance the proof. We have developed the system in HOL Light [1], and have used it in a non-trivial setting, namely incidence-reasoning for geometry theorem proving.

1 Introduction

Most interactive proof assistants are *user-driven*. Procedures and tactics are called upon to automatically build up a justification tree or construct a proof term. However, while the procedures can be combined in powerful ways, there is, as yet, no framework to build *discovery* tools. These could explore the proof-space independently and report back to the user, helping the user better understand the domain or solve the goal outright.

By working independently, such a framework can exploit the wasted processor cycles as the user works out the proof. To some extent, such idle-time has been exploited by Isabelle's [5] Sledgehammer [4] interface to run external first-order provers as background processes. However, the user does not have the same fine-grained control and composability of these tools as with tactics.

2 Framework

In Figure 1, we show how discovery can be integrated with theorem proving. Here, we are interested in *declarative* proof, where the hypotheses in the goal-state are just intermediate facts that reflect the proof written so far, and evolve to bring the system closer to the goal theorem. In our framework, these facts are generally pulled in by a primitive component in our discovery algebra called "Monitor". This primitive is composed with other primitives to yield a single domain-specific discoverer. The composite discoverer outputs to a fact database, whose contents can be applied by the user via a new declarative language primitive: `obviously`.

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 370–375, 2011.

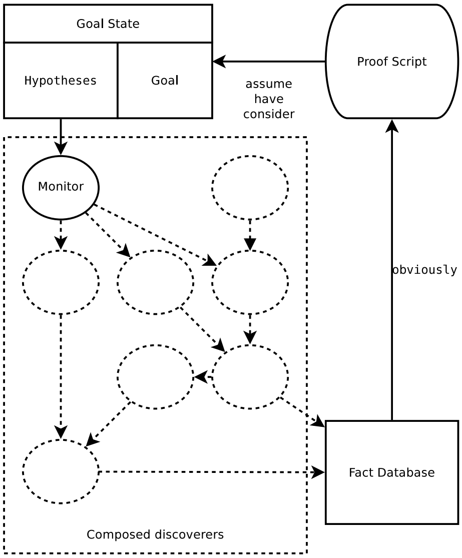


Fig. 1. Integration of Discovery with Interactive Proof

In this framework, the discovery tool and user *collaborate independently* in a feedback loop. By writing the proof script, the user is implicitly adding facts to the goal-state, helping direct the discovery. In turn, the discovery helps the user to understand the shape of the proof-space, and offers facts which can be used to write more of the proof. Moreover, the two systems run simultaneously, with the discoverer exploiting idle-time as the proof-script is written.

3 Composition

Ultimately, the discovery system outputs one or more theorems, which we can represent as a lazy list. We have favoured lists because of their ubiquity and support in functional programming languages such as Ocaml, a language whose top-level serves as the interface to the HOL Light theorem prover.

The key structure we identify on lists is a monad [9], which allows us to combine a list with a data-dependent list, to produce a single list. However, the standard list monad only works for finite search spaces, whereas the monad we consider here is intended for infinite streams. Its basic implementation is described in detail elsewhere [8], but the key part of the implementation is a *join* function, which converts a list of lists into a single concatenated list.

In our interpretation, a list is a stream of outputs produced by a discoverer, so the join function transforms a discoverer which outputs *discoverers* into a single flattened discoverer. This has a simple computational interpretation: each inner discoverer is *forked* where it appears in the output of the outer discoverer. Its results are then merged in parallel with all other forked discoverers.

This gives our basic algebra to describe theorem-discovery and combine theorem-discoverers. The way that discoverers are combined is much as it is for the list monad: all possible pairs of data are combined. But in theorem proving, we are often interested in how the space of facts partitions during case-analysis, so we choose to add a tree structure inside the streams. We can retain the overall monad structure if trees also constitute a monad and can be combined.

3.1 Trees

Our trees represent case-analyses. We give an example of such a tree in Figure 2, where we use Latin letters $(P, Q, R, \ldots)$ to represent branch labels, and Greek letters $(\phi, \psi, \chi, \ldots)$ to represent node formulas. The branch-labels identify the assumptions of case-analyses, while the nodes contain lists of facts inferred on the strength of the assumptions along the root path. So the tree in Figure 2 represents the formula:

$$\begin{aligned}\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n \wedge (P \rightarrow \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n) \wedge (Q \rightarrow \chi_1 \wedge \chi_2 \wedge \cdots \wedge \chi_n \\ \wedge (R \rightarrow \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n) \wedge (S \rightarrow \beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n))\end{aligned}$$

The principal operation on trees is a sum function which is analogous to the append function for lists, combining all values from two trees. We combine case-analyses by nesting them, replacing the leaf nodes of one tree with copies of the other tree. For definiteness, we always nest the right tree in the left.

We also simplify the resulting tree in the following ways: first, we close branches when there is no data in their subtrees; second, if data appears at the root of two subtrees, it is promoted into the parent node — a move which corresponds to disjunction elimination; third, if a case is introduced which has already been considered in a parallel branch further up the tree, it can be dropped; finally, if a branch label already appears on the root path, then its subtree can be merged into the parent — a move which corresponds to weakening. We illustrate these rules in Figure 3.

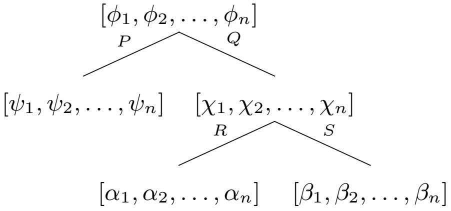

$[\phi_1, \phi_2, \ldots, \phi_n]$
P
Q
$[\psi_1, \psi_2, \ldots, \psi_n]$
$[\chi_1, \chi_2, \ldots, \chi_n]$
R
S
$[\alpha_1, \alpha_2, \ldots, \alpha_n]$
$[\beta_1, \beta_2, \ldots, \beta_n]$

Fig. 2. Tagged Proof-trees

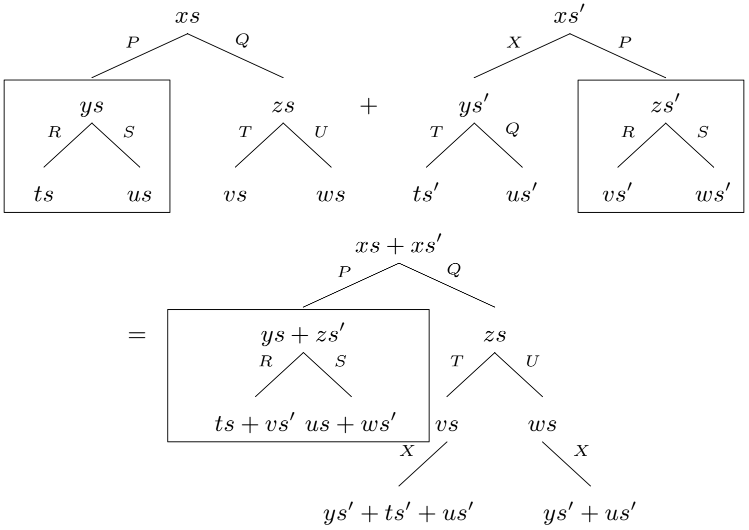


Fig. 3. Proof tree combination and simplification. The highlighted subtrees are combined with the most simplification, yielding a subtree with the same topology.

Primitive	Meaning
`merge` : α chain $\rightarrow$ α chain $\rightarrow$ α *chain*	Combine the results of two chains.
`null` : α chain	Identity of merge.
`iterate` : (α chain $\rightarrow$ α chain) $\rightarrow$ α chain $\rightarrow$ α chain	`iterate` $f x$ accumulates the results of x, $f\,x$, $f\,(f\,x)$, ...
`monitor` : thm chain	The chain whose elements are drawn from the goal state.
`gen` : term $\rightarrow$ thm chain $\rightarrow$ thm chain	Attempt to universally quantify a free variable across all facts in a chain.
`consider` : thm chain $\rightarrow$ thm chain	Convert existential facts to facts hypothesised on a witness.
`conjuncts` : thm chain $\rightarrow$ thm chain	Split conjuncts across a chain.
`disj_elim` : thm chain $\rightarrow$ thm chain $\rightarrow$ thm chain	Given a chain of disjunctions and a chain of cases-splits, perform disjunction elimination.
`rewrite` : conv chain $\rightarrow$ thm chain $\rightarrow$ thm chain	Use a chain of conversions to rewrite the facts in another chain.
`mp` : thm chain $\rightarrow$ thm chain $\rightarrow$ thm chain	Apply a chain of implications to a chain of antecedents.

Fig. 4. Some additional primitives and functions of chains

3.2 Chains

With the sum operation defined, we can define a natural join for trees, and thereby define a monad for trees. As we remarked in §3, this means that streams of these trees also constitute a monad, hereafter referred to as *chains*.

Applicative Functor. An applicative functor [3] is less expressive than a monad, in that it does not allow for data-driven search strategies [6]. However, we have a special implementation in which we can make stronger guarantees than the monad as to whether search has reached a fixpoint. If a user needs to detect fixpoints, and does not need search to be data-driven, then they might prefer to build a discoverer with combinators for the applicative functor.

Of these combinators, the main one allows us to apply a chain of functions to a chain of arguments, running all combinations to produce a chain of results. We can achieve this with the monad transformations, but with the applicative functor we have more flexibility in choosing when elements are combined. In our implementation, if f and x are the mth and nth elements of two chains, then the combined tree $f\,x$ will always appear at index $\max{(m,n)}$. Thus, if two chains become empty at the same index, the user can guarantee that no further combination is possible, and can stop drawing elements.

Primitives and Transformations. In addition to the basic algebra, we list some other useful chain transformations and primitives in Figure 4.

4 Results

We have implemented our chain language in HOL Light and integrated it with the Mizar Light declarative language [10]. We then used it to define a theorem-discoverer to help formalise Hilbert's *Foundations of Geometry* [2]. Our earlier formalisations of this text [7] showed that complex reasoning about unstated incidence properties dominates the proof text, particularly the existence of lines, triangles and planes which are needed to apply a complex axiom due to Pasch and reason about its disjunctive conclusion.

By using our algebra, we were able to define separate chains to discover triangles, lines and planes and represent their interdependencies via mutual recursion. A final chain applied Pasch's axiom, automatically performing a case-analysis on its conclusion.

The discoverer found interesting alternative ways to apply Pasch's axiom, including a completely alternative proof for Hilbert's fourth Theorem. In general, it reduced the number of formalised proof steps by a factor of 10.

5 Conclusion

We have outlined and implemented a declarative language for describing and composing concurrent discovery engines. The engines are integrated into a framework in which the discovery system collaborates with a user writing a forward declarative-style proof, exploiting wasted CPU cycles that arise when proofs are first formalised. Engines can be readily *composed* via a rich set of transformations which control search, case-splitting, existential reasoning and rewriting, allowing them to be crafted to handle specific domains. Indeed, we have a prototype engine which automates incidence reasoning in Hilbert's *Foundations of Geometry* [2].

For now, we have a straightforward interface, but we intend to build a more sophisticated customisable front-end for users. Since chains just produce lazy lists of facts, it is relatively easy to perform additional filtering and computation on the discovered results for presentation to the user.

As yet, the language does not provide any functions to perform lemma *speculation* of, say, inductive hypotheses. This, we believe, is not because the approach is *limited.* We have only avoided it because incidence-reasoning in *Foundations of Geometry* was our initial concern and only needs ground facts. The basic chain data-type defined is polymorphic and not specific to theorem-proving, so finding ways to *search* for speculative lemmas should just be a matter of *deriving* the appropriate chains. We hope to see this through with more case-studies.

Such case-studies will help us find new abstractions and derived transformations, and so make it easier to write chains. It will also help us investigate performance issues. As of now, there is a great deal of *rewriting* used by default that may cause a bottleneck in some searches. There are likely to be various optimisations we can make to improve this, even before we start thinking of enriching the underlying data-structures.

References

1. Harrison, J.: HOL Light: a Tutorial Introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
2. Hilbert, D.: Foundations of Geometry. Open Court Classics, 10th edn. (1971)
3. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1), 1–13 (2008)
4. Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: First prototype. Inf. Comput. 204(10), 1575–1596 (2006)
5. Paulson, L.C.: Isabelle: a Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
6. Sam Lindley, J.Y., Wadler, P.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. In: Mathematically Structured Functional Programming (2008), `http://hdl.handle.net/1842/3800`
7. Scott, P.: Mechanising Hilbert's Foundations of Geometry in Isabelle. Master's thesis, University of Edinburgh (2008)
8. Spivey, J.M.: Algebras for combinatorial search. Journal of Functional Programming 19(3-4), 469–487 (2009)
9. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, pp. 24–52 (1995)
10. Wiedijk, F.: Mizar Light for HOL Light. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 378–394. Springer, Heidelberg (2001)

Heterogeneous Proofs: Spider Diagrams Meet Higher-Order Provers

Matej Urbas and Mateja Jamnik

University of Cambridge
{Matej.Urbas,Mateja.Jamnik}@cl.cam.ac.uk

Abstract. We present an interactive heterogeneous theorem proving framework, which performs formal reasoning by arbitrarily mixing diagrammatic and sentential proof steps.

We use Isabelle to enable formal reasoning with either traditional sentences or spider diagrams. We provide a mechanisation of the theory of abstract spider diagrams and establish a formal link between diagrammatic concepts and the existing theories in Isabelle/HOL.

1 Introduction

Diagrams are often employed as illustrations in "pen and paper" reasoning. In the past, they frequently formed essential parts of proofs. Eventually, with advent of proof theory, their role became almost exclusively that of a visual help. Still, the intuitive nature of diagrams motivated the design of formal diagrammatic reasoning systems – for example, spider diagrams [6] and constraint diagrams [3]. Consequently, some purely diagrammatic theorem provers have been developed, Diamond [8], Edith [10] and Dr.Doodle [13] are some examples.

Heterogeneous reasoning was the next step in the development of diagrammatic reasoning systems. It merged the diagrammatic and sentential modes of reasoning and allowed proof steps to be applied to either diagrammatic, sentential or mixed formulae. In the paper *Reasoning with Sentences and Diagrams* [5], Hammer laid the formal foundations for such heterogeneous reasoning systems.

Later, Barwise [2], Barker-Plummer [1] and Shin [9] investigated heterogeneous reasoning software. The result was a framework called Openproof [1], which uses diagrammatic representation as an input method. The diagrammatic part of the framework is not formalised within the logic of the sentential reasoner. Therefore, the diagrammatic and sentential components remain logically separated.

Our goal is to enable formal interactive heterogeneous reasoning in a general purpose theorem prover. We investigate three aspects of interactive heterogeneous reasoning: *a)* the direction of proofs (e.g., from a diagrammatic assumption to a sentential conclusion and vice versa), *b)* expression of statements that contain mixed sentential and diagrammatic terms, and *c)* mixed application of diagrammatic, sentential, and heterogeneous inference steps.

Our key motivation is to provide different points of view on formulae and to enable reasoning about diagrams sententially or vice versa. We believe that

M. Van Eekelen et al. (Eds.): ITP 2011, LNCS 6898, pp. 376–382, 2011.

heterogeneous reasoning will not only serve as a pedagogical tool for introduction to logic, it may also improve intuitiveness and readability of formulae (and proofs) in specific domains of verification – analogous to other domain specific languages. Another motivation for heterogeneous reasoning is the ability to augment diagrams with sentential reasoning wherever diagrams fall short.

In contrast to the approach of Openproof our aim is not to keep the two reasoning systems separated, but to integrate them using heterogeneous representation and reasoning. In addition, we want to formalise diagrams and some of their inference rules in the logic of an LCF-style [4] higher-order theorem prover. With this we aim to enable certified proof reconstruction of heterogeneous proofs.

In order to build a heterogeneous reasoning framework, we first chose an existing diagrammatic reasoning language called *spider diagrams* (see Section 2). The second part is the sentential reasoner, for which we chose Isabelle [12].

We formalised the theory of spider diagrams in Isabelle/HOL (see Section 3.1). This enabled sentential reasoning about diagrams and also simplified translation from spider diagrams to sentences (see Section 3.2). Translation from sentences to diagrams, proof automation, and proof reconstruction, however, is still work in progress. Diagrammatic reasoning will be done in Speedith, our own external reasoner, which is currently in development (see Section 4).

2 The Diagrammatic Language

We have picked the language of spider diagrams [7] as the diagrammatic part of our heterogeneous reasoner because it has a formally defined syntax and semantics. Spider diagrams are equivalent to first-order monadic logic and are equipped with a number of purely diagrammatic inference rules, which have been shown to be sound[1].

Spider diagrams consist of the following basic elements (see Figure 1):

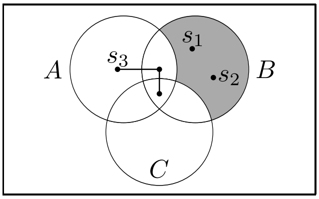


Fig. 1. A spider diagram featuring all the basic diagrammatic language elements

Contours represent named sets. They are drawn as labelled circles (e.g., circles A, B and C in Figure 1).

Zones are also outlined areas and denote specific subsets of contours and their complements (Figure 1 contains 8 zones).

Spiders are single existentially quantified elements. One spider is a finite collection of dots that are connected with lines. The dots are called *feet*, which indicate the zones in which the spider may live.

Shaded zones indicate that a zone is a subset of its spiders (i.e., the set this zone represents may contain only spiders with a foot in it).

[1] For more detail see *Spider Diagrams* [7] by Howse et al, and *The expressiveness of spider diagrams augmented with constants* [11] by Stapleton et al.

Contours and zones are both outlined shapes representing sets. Contours are labelled with alphabetical letters. Zones are not labelled and represent intersections and complements of contours.

Zones are defined as ordered pairs, say $Z = (\mathit{\Gamma},\ \mathit{\Delta})$. The first element of the pair, $\mathit{\Gamma}$, is a set of contours which contain the zone. The second element, here $\mathit{\Delta}$, is a set of contours which **do not** contain the zone (the zone lies entirely outside of these). The set described by the zone $(\mathit{\Gamma},\ \mathit{\Delta})$ is defined as

$$\texttt{set\_of}(Z) = \bigcap_{A_i \in \mathit{\Gamma}} \texttt{set\_of}(A_i) \setminus \bigcup_{B_i \in \mathit{\Delta}} \texttt{set\_of}(B_i), \tag{1}$$

where $\texttt{set\_of}(A_i)$ is the set represented by countour A_i. For example, the spider diagram in Figure 1 contains 8 zones (note that zone $(\{\}, \{A, B, C\})$ lies outside all contours). However, not all zones have to be drawn. They may be omitted if they play no role in the statement.

Spiders represent single elements. Spiders are dots which may optionally be connected with a line. Dots are the *feet* of the spider and define its *habitat.* As an example, Figure 1 contains three spiders: spiders s_1 and s_2 reside in zone $(\{B\}, \{A, C\})$, spider s_3 resides in a *region* of three zones. Regions are collections of zones, with corresponding sets defined as follows:

$$\texttt{set\_of}(R) = \bigcup_{Z_i \in R} \texttt{set\_of}(Z_i) \tag{2}$$

The following is an illustration of the semantics of the diagram in Figure 1:

$$\begin{array}{c} \exists s_1\, s_2\, s_3.\ \mathit{distinct}(s_1, s_2, s_3) \wedge (s_1 \in B \setminus A \cup C) \wedge (s_2 \in B \setminus A \cup C)\ \wedge \\ (s_3 \in (A \setminus C) \cup (A \cap B \cap C)) \wedge (B \setminus A \cup C) \subseteq \{s_1, s_2\} \end{array} \tag{3}$$

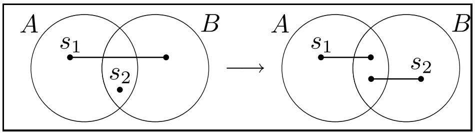


Fig. 2. A diagrammatic statement in the language of spider diagrams

A *compound spider diagram* is a diagram that consists of spider diagrams, which are called *unitary spider diagrams*, coupled with logical connectives. Figure 2 is an example of a compound spider diagram. Formula 4 illustrates the semantics of the diagram in Figure 2:

$$\begin{array}{c} \exists s_1\, s_2.\ \mathit{distinct}(s_1, s_2) \wedge (s_1 \in A \cup B \setminus A \cap B) \wedge (s_2 \in A \cap B) \\ \longrightarrow \\ \exists s_1\, s_2.\ (s_1 \in A) \wedge (s_2 \in B) \end{array} \tag{4}$$

Note that spider names are local to unitary spider diagrams, whereas contour names are global.

Figure 3 shows a purely diagrammatic proof of the example in Figure 2. Note that the proof involves applications of three sound diagrammatic inference rules

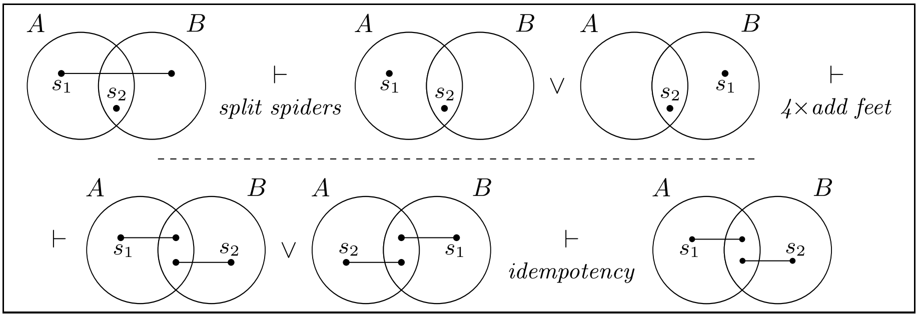


Fig. 3. A purely diagrammatic proof of the example in Figure 2

(from [7]): split spiders (removes lines connecting feet of a spider and creates a case-split for each foot), add feet (puts a new dot into a zone and connects it to an existing spider in a foreign region), and idempotency. Our goal is to enable mixing of these and other diagrammatic inference rules with sentential ones.

3 Sentential Reasoner

Our first step was to provide a formalisation of the theory of spider diagrams within Isabelle/HOL (files available from `http://gitorious.net/speedith`). This not only makes the translation between the two representations easier, but also allows for direct proof reconstruction within Isabelle.

3.1 Formalisation of Diagrams in Isabelle/HOL

We formalise the basic elements of spider diagrams as follows:

Contours are identifiers of type *`contour = nat`* (natural numbers).

Zones are sets of contours (sets of natural numbers). Internally, zones are of the following type: *`zone = contour set`* (or equivalently *`zone = nat set`*).

Regions are sets of zones: *`region = zone set`* (or equivalently *`region = (nat set) set`*).

Spiders are identifiers of type *`spider = nat`*.

The interpretation of each of the above diagrammatic elements is provided by their corresponding *mapping functions*. These functions take the above identifiers and return sets (or elements – for spiders) that correspond to them. Figure 4 shows the map function for zones:

```
fun zmap :: "('e, 'a) SD_scheme ⇒ zone ⇒ 'e set" where
  "zmap_d cs = (⋂c ∈ cs. cmap_d c) - (⋃c ∈ (-cs). cmap_d c)"
```

Fig. 4. The definition of the zone map which maps a zone to its set

```
lemma add_feet: "⟦ smap s ∈ rmap r; r ⊆ r' ⟧ ⟹ smap s ∈ rmap r'"
```

Fig. 5. The sentential equivalent of the *add feet* diagrammatic inference rule

```
lemma: "(∃s s'. s ≠ s' ∧ smap s' ∈ rmap {{0}, {1}} ∧ smap s ∈ rmap {{ 0, 1 }}) ⟶
(∃s s'. s ≠ s' ∧ smap s ∈ rmap {{0}, {0, 1}} ∧ smap s' ∈ rmap {{1}, {0, 1}})"
```

Fig. 6. A sentential translation of the example in Figure 2

We also provide proofs for relevant lemmas of the theory of abstract spider diagrams (from [7]), e.g.: disjointness of zones, additivity of the region map over unions, intersections and complements. In addition to these, we have also formalised the diagrammatic inference rules mentioned above (i.e., split spiders, add feet and idempotency). Figure 5 shows the formalised *add feet* rule.

3.2 Translation

The lemma in Figure 6 is the sentential translation of the diagrammatic statement in Figure 2.

Translation from diagrams to sentences currently generates n existentially quantified first-order variables, a conjunction of $\frac{n(n-1)}{2}$ inequalities and n set-inclusion predicates, where n is the number of spiders. Using the higher-order quantification provided in Isabelle/HOL, we can existentially quantify over a single set of spiders. With a single predicate, say `distinct`, we can also remove the inequalities and make translation to diagrams easier. We aim to translate as many MFOL formulae to diagrams as possible.

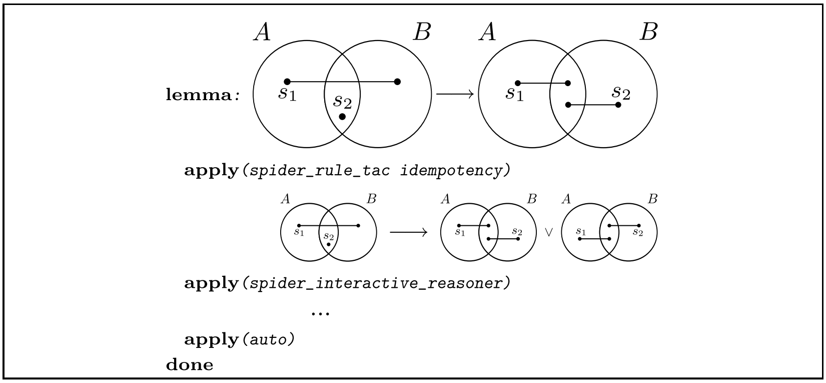
Heterogeneous proof automation, proof reconstruction, and translation of sentences to diagrammatic representation is work in progress.

4 Heterogeneous Integration

Ultimately, we want to enable formal graphical reasoning as is depicted in Figure 7. Sentential expressions are drawn as diagrams if the translation is feasible. More importantly, the external diagrammatic reasoner can be invoked in an interactive mode within the proof body like any other tactic in Isabelle. These tactics will invoke our diagrammatic reasoner, which in turn will return a proof trace for proof reconstruction.

We also want to enable visual and interactive diagram construction and application of diagrammatic inference steps. For this purpose we will use Speedith (sources available from `http://gitorious.net/speedith`) both as a standalone reasoning tool as well as a visual add-on to Isabelle's graphical user interfaces.

The user will be able to invoke the diagrammatic reasoner at any time during the proof, with an option to do so in an interactive or fully automated mode. Additionally, the currently active statement (lemma or proof obligation) will be automatically visualised as a diagram in an embedded window of the GUI.



```
lemma:  ⟶
  apply(spider_rule_tac idempotency)
  ⟶  ∨
  apply(spider_interactive_reasoner)
  ...
  apply(auto)
done
```

Fig. 7. A heterogeneous proof outline of the example in Figure 2

Discussion. We outline a work-in-progress of an integration of a diagrammatic language with diagrammatic inference rules into a sentential theorem prover. This enables formal heterogeneous reasoning with mixed diagrams and sentences.

We provide a formalisation of spider diagrams with translation to sentential formulae. Heterogeneous proof automation, and a translation from first-order monadic formulae to spider diagrams is still work in progress.

The goal of this project is to provide a proof-of-concept heterogeneous reasoner – to show that heterogeneous reasoning is feasible. Ultimately, we plan to extend the heterogeneous framework to other domains with new diagrammatic systems (e.g.: constraint diagrams, UML, diagram chasing etc.).

In summary, we believe that heterogeneous reasoning can improve the ease of working with specific domains of verification in general purpose theorem provers.

Acknowledgments. We would like to thank Thomas Tuerk and Lawrence Paulson for invaluable input.

References

1. Barker-Plummer, D., Etchemendy, J., Liu, A., Murray, M., Swoboda, N.: Openproof - A Flexible Framework for Heterogeneous Reasoning. In: Stapleton, G., Howse, J., Lee, J. (eds.) Diagrams 2008. LNCS (LNAI), vol. 5223, pp. 347–349. Springer, Heidelberg (2008)
2. Barwise, J., Etchemendy, J.: A Computational Architecture for Heterogeneous Reasoning. In: TARK, pp. 1–11. Morgan Kaufmann, San Francisco (1998)
3. Gil, J., Howse, J., Kent, S.: Towards a Formalization of Constraint Diagrams. In: IEEE Symposia on Human-Centric Computing Languages and Environments, p. 72 (2001)
4. Gordon, M., Wadsworth, C.P., Milner, R.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78. Springer, Heidelberg (1979)
5. Hammer, E.: Reasoning with Sentences and Diagrams. NDJFL 35(1), 73–87 (1994)
6. Howse, J., Molina, F., Taylor, J., Kent, S., Gil, J.: Spider Diagrams: A Diagrammatic Reasoning System. JVLC 12(3), 299–324 (2001)

7. Howse, J., Stapleton, G., Taylor, J.: Spider Diagrams. LMS JCM 8, 145–194 (2005)
8. Jamnik, M., Bundy, A., Green, I.: On Automating Diagrammatic Proofs of Arithmetic Arguments. JOLLI 8(3), 297–321 (1999)
9. Shin, S.-J.: Heterogeneous Reasoning and its Logic. BSL 10(1), 86–106 (2004)
10. Stapleton, G., Masthoff, J., Flower, J., Fish, A., Southern, J.: Automated Theorem Proving in Euler Diagram Systems. JAR 39(4), 431–470 (2007)
11. Stapleton, G., Taylor, J., Thompson, S., Howse, J.: The expressiveness of spider diagrams augmented with constants. JVLC 20(1), 30 (2009)
12. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)
13. Winterstein, D., Bundy, A., Gurr, C.: Dr.Doodle: A diagrammatic theorem prover. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 331–335. Springer, Heidelberg (2004)

Author Index