Emerging Trends
Proceedings of the 17th International Conference
on Theorem Proving in Higher Order Logics:
TPHOLs 2004

Preface

This volume constitutes the proceedings of the Emerging Trends track of the *17th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs 2004) held September 14-17, 2004 in Park City, Utah, USA. The TPHOLs conference covers all aspects of theorem proving in higher order logics as well as related topics in theorem proving and verification.

There were 42 papers submitted to TPHOLs 2004 in the full research category, each of which was refereed by at least 3 reviewers selected by the program committee. Of these submissions, 21 were accepted for presentation at the conference and publication in volume 3223 of Springer's *Lecture Notes in Computer Science* series.

In keeping with longstanding tradition, TPHOLs 2004 also offered a venue for the presentation of work in progress, where researchers invite discussion by means of a brief introductory talk and then discuss their work at a poster session. The work-in-progress papers are held in this volume, which is published as a 2004 technical report of the School of Computing at the University of Utah.

August 2004 *Konrad Slind*

Contents

Some Mathematical Case Studies in ProofPower-HOL 1
R. D. Arthan

A Framework for Interactive Sharing and
Deductive Searching in Distributed Heterogeneous
Collections of Formalized Mathematics 17
James L. Caldwell and Christoph Jechlitschek

Mechanical Verification of Automatic Synthesis of
Failsafe Fault-Tolerance 35
Sandeep S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir

ARM6 Formal Verification:
Experience with a Commercial Microprocessor 47
Anthony Fox

Building Extensible Compilers in a Formal Framework:
A Formal Framework User's Perspective 57
Nathaniel Gray, Jason Hickey, Aleksey Nogin, and Cristian Ţăpuş

Compiling HOL4 to Native Code 71
Joe Hurd

Higher-Level Hardware Synthesis in HOL 79
Juliano Iyoda and Michael J.C. Gordon

An Experiment in Automated Theorem Proving in Type Theory . . . 95
Marcin Benke and Fredrik Lindblad

Cooperating Theorem Provers:
A Case Study Combining HOL Light and CVS Light 109
Sean McLaughlin and Clark Barrett

Embedding Multiway Decision Graphs in HOL 121
Tarek Mhamdi and Sofiéne Tahar

Formalizing the AMBA High Performance Bus 137
Malcolm C. Newey

Implementing the Calculus of Inductive Constructions in the
MetaPRL Framework 153
Natalia Novak and Yegor Bryukhov

Towards Verified Virtual Memory in L4 165
Gerwin Klein and Harvey Tuch

XPath Formal Semantics and Beyond: a Coq based approach 181
Pierre Genevés and Jean-Yves Vion-Dury

The Axiomatization of Group Theory:
An Experiment in Constructive Set Theory 199
Xin Yu and Jason Hickey

Some Mathematical Case Studies in ProofPower-HOL

R.D. Arthan

Lemma 1 Ltd.
2nd Floor, 31A Chain Street,
Reading UK RG1 2HX
rda@lemma-one.com

Abstract. This paper gives an overview of three case studies in developing pure mathematical theory using ProofPower-HOL. The case studies, which currently cover a selection of basic material from the theories of real analysis, group theory and topology, expose some interesting issues for formalising mathematics.

1 Introduction

Apart from basic mathematical structures such as sets, functions, lists and numbers, applying an automated theorem-proving system to hardware and software engineering problems tends to involve mathematical theories of a rather different nature from the traditional subject matter of pure mathematics. Many researchers feel that engineering applications are the most important for automated theorem-proving. However, it is natural to try to formalise pure mathematical theories. Research into this goes back to the earliest days of electronic computation.

In a recent survey, Carlos Simpson [12] has identified numerous reasons why computer-assisted formalised mathematics should be of benefit to the mathematical community. Simpson gives many references to earlier work in this area as does John Harrison in his thesis [7] and his paper [6]. The Flyspeck project [5] is applying computer-assisted theorem proving to increase confidence in Thomas Hales' proof of the Kepler sphere-packing conjecture, a difficult proof involving a considerable element of computation which has caused problems for the traditional peer review process.

In 2001, the opportunity arose to develop, for use in program verification, a theory of real arithmetic for the ProofPower specification and proof system. It was a natural experiment to use this as the basis of a theory of real analysis and I spent some time late in 2001 working on that. In 2003, since the Jordan curve theorem[1] was felt to be a good challenge problem in some automated theorem-proving circles, I used ProofPower-HOL to prove what Henle [8] calls the

[1] Apparently, much progress has been made on the Jordan curve theorem using the Mizar system, but I not able to assess whether the proof of the general case in two dimensions is complete (see http://mizar.uwb.edu.pl/).

fundamental lemma in one of the classical proofs of this results. In retrospect, I view this as a highly instructive mistake: Henle's book is a very accessible account of elementary algebraic topology for beginning students. His fundamental lemma is essentially a calculation of the mod 2 homology groups of the plane. I formulated it as a combinatorial result about discrete gratings and proved it, the proof being fairly easy.

Unfortunately, connecting the fundamental lemma expressed as a combinatorial fact with the geometry involves several topological results, most notably what Henle calls Alexander's lemma. I quickly realised that I was going to have to cover quite a bit of geometry and topology to prove them. Now Henle's proofs are very carefully designed for the beginner; they appeal to geometric intuitions as much as to formal reasoning. Henle sets up topology as the topology of subsets of the plane and to follow his proofs as they stand would involve doing special cases of general results whose proofs are no harder formally than the special cases.

Moral 1: if you ask someone "have you proved the XYZ theorem?" and receive the reply that they have proved the "fundamental lemma" or the "main result" or similar, it is wise to scrutinise their formal account closely to find out what they have actually proved[2].

Moral 2: theorem-provers don't need spoon-feeding; it makes sense to prove things at the "right" level of generality and that will often be more general than in an account intended for beginners.

Moral 3: while there is no royal road to proving theorems, there are shortcuts; however, you have to choose your shortcuts very carefully to make sure you don't get lost.

I subsequently began some case studies in pure mathematics, trying to cover the material along the lines that it might be covered in a typical undergraduate or beginning graduate course. Carried far enough, this programme would have the Jordan curve theorem drop out as the two dimensional case of the Jordan-Brouwer separation theorem proved via the calculation of the homology groups of spheres, but that is a long way off. To date this work has covered the following topics.

- A more complete treatment of real analysis including the definitions and basic properties of the exponential and trigonometric functions and of π.
- Some group theory including the definitions and elementary properties up to the three isomorphism theorems and the Cayley representation theorem
- Enough abstract and metric space topology to define the notions of homotopy and the fundamental groupoid (and to prove that it is a groupoid).

[2] As a simpler example, I have still not seen a proof of the mutilated chess-board theorem as a theorem about dominoes and chess-board as geometrical objects, which is what they surely are. Just as in my problem with the Jordan curve theorem, the combinatorics is fairly easy, but the geometric realisation requires more work.

My objectives in this enterprise were somewhat vague: essentially, I just wanted to see how this material turns out and to compare notes[3] with other systems (Mizar, PVS, HOL Light, etc.). I was also specifically interested in developing the theory to the points where the main mathematical subjects of algebra, analysis, geometry and topology begin to interact and inform one another, e.g., in algebraic topology and differential geometry. During the course of the work, some definite themes have emerged:

- I have tried to provide natural and readable specifications of the mathematical concepts formalised. For example, I use differential equations rather than power series as the definitions of the trigonometric functions, since I consider that approach to have a more intuitive, geometrical appeal.
- I have tried to follow the development of pure mathematics both in fitting abstract notions to more concrete ones after the event and in using abstract notions that have not yet been formalised to inform more concrete work. For example, you don't need to develop abstract group theory to define the real numbers and show that they are a group under addition. You can even use group-theoretical thinking while you're developing the theory. However, once you have some abstract group theory, you should be able to apply that to the real numbers and other specific constructions you may make with them (e.g., real vector spaces[4]).
- I have tried to develop each theory at the "right" level of generality or abstraction: this often involves a compromise between making the task at hand feasible and making the results general enough to be useful. On the other hand, being more abstract is sometimes both more powerful in applications and easier! E.g., the fundamental groupoid of a topological space is technically often easier to work with than the fundamental group.

There is no new mathematics of any significance in any of this: just as there is no significant new mathematics in an undergraduate textbook. However, interesting details arise *en passant* and you do learn something as you go (for example, that integration is not needed to develop enough of the the theory of power series to introduce the exponential and transcendental functions).

This paper gives an overview of what has been done at the time of writing (May–July 2004) and discusses some of the issues for formalising mathematics that have been highlighted. Full details are given in the papers [1–3]. The structure of the sequel is as follows: section 2 introduces the ProofPower-HOL logic and system by means of a simple example which also illustrates, in microcosm, some of the formalisation issues encountered (a theory listing for this example is given as an appendix); section 3 gives an outline on what has been done in the three case studies; section 4 discusses how the approach of the case studies

[3] The present paper is not intended as a detailed presentation of comparisons between different systems, but I will air my personal viewss on some points of principle.

[4] Roger Jones and I have an embryonic theory of normed real vector spaces based on the group theory case study and including a coordinate-free definition of the Frechet derivative.

might scale to more complex problem domains; section 5 gives some concluding remarks.

2 An Example

ProofPower is a system supporting specification and proof in HOL and Z. It is founded on an LCF-style implementation in Standard ML[5] of the same polymorphic simple type theory as the other systems in the HOL family. ProofPower-HOL supports a syntax for specification adapted from the Z notation [13] intended to encourage well-documented formal specifications using familiar logical and mathematical notations. This document is written in that syntax, Its source form is a mixture of LaTeX and input for the ProofPower-HOL parser. The input for the parser is displayed using a special font and a (mostly) single-character mark-up for the mathematical symbols, so that, for example, when you see '$\forall$' in this paper, what I typed and then saw on my screen was an upside-down 'A' too.

To illustrate the style of specification adopted in the case studies, let us develop a simple algebraic theory. If G is a group, a *G-action* on a set X is a correspondence between elements of G and mappings of X to itself such that multiplication in the group corresponds to composition of the mappings and the unit of the group corresponds to the identity mapping. A set X equipped with a G-action is called a G-set. G-sets arise, for example, by considering groups of symmetries of geometrical objects. To give a concrete example of a group action before defining the concept of a group, let us consider the particular case when G is $\mathbb{Z}$, the group of integers under addition.

So a $\mathbb{Z}$-set will comprise a pair comprising a set (called the *carrier set* of the $\mathbb{Z}$-set) together with an assignment to each integer of a function from the set to itself. To represent this abstract concept in HOL, let us consider the polymorphic class of all pairs comprising a set of elements of some type $'a$ together with a function mapping integers to total functions from $'a$ to itself. We can capture this in the following type abbreviation[6]

SML

$$declare\_type\_abbrev(\texttt{"}\mathbb{Z}\_SET\texttt{"},\ [\texttt{"}'a\texttt{"}],\ \ulcorner :'a\ SET \times (\mathbb{Z} \to 'a \to 'a)\urcorner);$$

The type $'a$ here is a polymorphic type parameter. It can be instantiated to any type we please, for example, an element of the type[7] $\mathbb{R}\ \mathbb{Z}\_SET$ is the type that includes all $\mathbb{Z}$-actions on sets of real numbers. We will think of the above

[5] ML stands for "metalanguage". Standard ML is a functional programming language which serves as both the implementation language and the interactive command language for ProofPower.

[6] Here the "specification" comprises an ML command to achieve the desired effect, since the ProofPower-HOL parser does not provide a concrete syntax for this form of definition.

[7] HOL type constructors are generally postfix operators, for example '$\mathbb{Z}$ LIST' denotes the type of lists of integers.

type as providing a *signature* for a class of *structures* which are candidates to be $\mathbb{Z}$-sets. If X is such a structure (i.e., a member of an instance of the above type), we will write $Car\ X$ for the carrier set and $(x\ **\ i)X$ for the action of an integer i on an element x. Note that the action operation is ternary not binary: in informal mathematics, it is normal to let the reader infer from the context which mathematical structures are being deployed, but formally we must be explicit about this.

To achieve the above syntax, we first declare the string '$**$' to act as an infix symbol with the same numerical precedence (310) as arithmetic exponentiation.

SML

```
declare_infix(310, "**");
```

We now give a constant specification to introduce the new constants 'Car' and '$**$'. A constant specification in ProofPower-HOL comprises two parts: the part above the line gives type ascriptions for the new constant or constants and the part below the line gives a predicate which is to be their defining property. In this case the defining property comprises two universally quantified equations defining the values of applications of the functions 'Car' and '$**$' Parsing the constant specification maps onto a call of the primitive definitional principle *const_spec*. This principle requires an existence proof for the constants being introduced. The ProofPower-HOL infrastructure includes a range of procedures for discharging the existence proofs and these will automatically discharge the proof obligations for all of the definitions in this example.

HOL Constant

$$
\begin{array}{l}
\boldsymbol{Car} : \ 'a\ \mathbb{Z}\_SET \rightarrow 'a\ SET; \\
\$\boldsymbol{**} : \ 'a \rightarrow \mathbb{Z} \rightarrow 'a\ \mathbb{Z}\_SET \rightarrow 'a \\
\hline
\forall\ (set,\ action)\bullet \\
\quad Car\ (set,\ action) = set \\
\wedge\ (\forall x\ i\bullet\ (x\ **\ i)\ (set,\ action) = action\ i\ x)
\end{array}
$$

The above definition serves to provide a convenient syntax for the operations on the structures of interest. We can see this in the following definition which captures the laws that a candidate $\mathbb{Z}$-set must satisfy to be worthy of the name. The laws specify that: *(i)*, the carrier set is closed under the $\mathbb{Z}$-action; *(ii)*, addition of integers corresponds to composition of the corresponding actions; and, *(iii)*, 0 corresponds to the identity function.

HOL Constant

$$
\begin{array}{l}
\mathbb{Z}\_\boldsymbol{Set} : 'a\ \mathbb{Z}\_SET\ SET \\
\hline
\forall\ X\bullet \\
\quad X \in\ \mathbb{Z}\_Set \\
\Leftrightarrow(\forall x\ i\bullet\ x \in Car\ X \Rightarrow (x\ **\ i)\ X \in Car\ X)
\end{array}
$$

$$\wedge\ (\forall x\ i\ j \bullet\ x \in Car\ X \Rightarrow (x\ **\ (i + j))\ X = ((x\ **\ i)\ X\ **\ j)\ X)$$
$$\wedge\ (\forall x \bullet\ x \in Car\ X \Rightarrow\ (x\ **\ \mathbb{NZ}\ 0)\ X = x)$$

In addition to the specifications, the source of this document also contains the statements and proofs of a small selection of theorems about $\mathbb{Z}$-sets. ML proof scripts are not particularly informative even to the expert eye, except when they are brought alive by replaying them interactively, so they have been suppressed from the printed form of this document. There is a listing of the theory in the appendix. The reader is invited to refer to the appendix for the statements of the following two theorems which are both elementary consequences of the above definition. The first theorem says that acting by i and then by $-i$ results in the identity function on the carrier set and the second gives a cancellation law.

$\mathbb{Z}\_set\_minus\_thm$ $\mathbb{Z}\_set\_cancel\_thm$

We complete the example by defining the *orbit* of an element x of a $\mathbb{Z}$-set X. The orbit comprises the set of all elements y that can be reached from x under the $\mathbb{Z}$-action.

HOL Constant

$$\boldsymbol{Orbit} : 'a\ \mathbb{Z}\_SET \rightarrow 'a \rightarrow 'a\ SET$$

$$\forall\ X\ x \bullet\ Orbit\ X\ x = \{y \mid \exists i \bullet\ y = (x\ **\ i)\ X\}$$

The reader may again consult the appendix for the statements of the following two theorems. The first says that any element of a $\mathbb{Z}$-set belongs to its own orbit and the second says that any two orbits are either equal or disjoint. In other words, the orbits are the equivalence classes of an equivalence relation: "co-orbital".

$orbit\_refl\_thm$ $orbit\_disjoint\_thm$

In this example, we have formalised a very elementary mathematical theory and developed some very elementary theorems about it. The proofs would serve almost as they stand to prove the same facts about G-sets for arbitrary groups G given the theory of groups developed in [2]. This could then provide the basis of some much more interesting mathematics. For present purposes, the example serves to illustrate ProofPower in action and to introduce the style of presentation of analysis, topology and group theory in [1–3].

3 The Case Studies

3.1 Basic Analysis

The case study on analysis is presented in [1]. It builds on the ProofPower-HOL theory that introduces the real numbers as a complete ordered field and covers the following ground.

– polynomial functions on the real numbers
– limits of sequences of real numbers
– continuity of functions
– differentiation
– limits of function values
– uniform convergence of limits of functions
– series and power series
– special functions: exponential function, natural logarithm, sine and cosine.

Broadly similar subject matter has been formalised before in HOL Light by John Harrison [7] and by Hanne Gottliebsen [4] in PVS. There are also developments of analysis in Mizar and Coq and several other systems. In addition to proofs of theorems, the ProofPower treatment includes automated proof procedures for continuity-checking and calculating derivatives (as do the treatments of Harrison and Gottliebsen).

While I make no claim for novelty in the material covered, I would claim that the specifications are readable and natural and that the material that is covered is done comprehensively. For example, here are the definitions of the sin and cos functions and of Archimedes' constant, π, defined here as the positive generator of the additive group of roots of the sin function.

$$\boldsymbol{Sin\ Cos} : \mathbb{R} \rightarrow \mathbb{R}$$

$$Sin(\mathbb{NR}\ 0) = \mathbb{NR}\ 0 \land Cos(\mathbb{NR}\ 0) = \mathbb{NR}\ 1$$
$$\land\ (\forall x \bullet\ (Sin\ Deriv\ Cos\ x)\ x) \land (\forall x \bullet\ (Cos\ Deriv \sim(Sin\ x))\ x)$$

$$\boldsymbol{ArchimedesConstant} : \mathbb{R}$$

$$\mathbb{NR}\ 0 < ArchimedesConstant$$
$$\land\ Sin(ArchimedesConstant) = \mathbb{NR}\ 0$$
$$\land\ (\forall x \bullet\ Sin\ x = \mathbb{NR}\ 0$$
$$\Rightarrow\ (\exists m \bullet\ x = \mathbb{NR}\ m * ArchimedesConstant)$$
$$\lor\ \ (\exists m \bullet\ x = \sim(\mathbb{NR}\ m * ArchimedesConstant)))$$

$$declare\_alias(\texttt{"}\pi\texttt{"}, \ulcorner ArchimedesConstant \urcorner);$$

Here the notation $(f\ Deriv\ c)\ x$ means that function f has derivative c at x and the function $\mathbb{NR}$ is the injection of the natural numbers into the reals. The alias declaration introduces the traditional name "π" as an alternative to "*ArchimedesConstant*".

The specifications of the trigonometric functions and of π clearly require non-trivial consistency proofs. This involves a development of the theory of power series, including the general result on differentiating power series term-by-term

(which avoids the need for introducing integration at this stage). The elementary properties of the exponential, logarithmic and trigonometric functions and π are then developed "axiomatically" from the differential equations.

As observed in [7], several notions of limit arise and it is desirable to have common ways of dealing with them. Harrison's approach is via the general notion of convergence nets. I use the more homely device of reducing the notions in question to sequential convergence. For example, it is an easy consequence of the standard definition of continuity that a function f is continuous at x iff. f maps any sequence converging to x to a sequence converging to $f(x)$. Using this fact, statements about continuity reduce to statements about sequential convergence, and, by and large, this turns the $\forall\exists\forall$ quantifier structure of the usual ϵ-δ arguments into simple universally quantified statements about sequential convergence. Proponents of non-standard analysis both in education and in theorem-proving sometimes advocate the simple quantifier structure of the definition of continuity in non-standard analysis as an advantage. Using sequential convergence achieves much the same effect in standard analysis. The text books tend not to stress this method of working if they mention it at all, probably because it fails to generalise to arbitrary topological spaces.

Moral 4: when you are using a theorem-prover, you do not need to adopt methods for their pedagogical value: unlike a student, the prover cannot develop bad habits, so you can freely use any method that works.

3.2 Group Theory

The case study on analysis deals with a single specific HOL type: the type $\mathbb{R}$ of real numbers. The case study [2] on group theory puts the polymorphism in HOL to work along much the same lines as the $\mathbb{Z}$-set example presented in section 2 above.

The case study begins with a treatment of equivalence relations, equivalence classes and the construction of quotient sets along the lines proposed by Larry Paulson [10]. This material comprises a lemma library which provides templates for working with equivalence relations, in particular, for defining functions on quotient sets. This supports the proof of the first isomorphism theorem in group theory, which is all about defining homomorphisms on quotient groups. It would serve a similar purpose in any of the common algebraic concrete categories (rings, modules over a ring, vector spaces over a field etc.) and in dealing with quotient spaces in topology.

The group theory itself begins with a definition of the signature of a group along similar lines to the signature for $\mathbb{Z}$-sets in the example above. The polymorphic notion of a group is then defined to be the set of all structures with this signature that satisfy the group laws.

Substructures and quotient structures in algebra are very important, so it is vital to deal smoothly with subgroups and quotient groups. Taken verbatim, the traditional explication of these concepts in set theory leads to significant notational and semantic difficulties. The problem is this: in doing the general theory, an expression like $x.y$ denoting the product of two elements of a group

G actually contains three variables: the group elements 'x', 'y', and the multiplication operator '.'. Syntactic tricks allow one to preserve something like the traditional infix notation for such expressions. But there is a semantic problem when one needs to deal with subgroups: according to the traditional account, the '.', in $x.y$ will denote a different set-theoretic function in a subgroup H from what it does in the containing group G. Coercing operations from subgroup to containing group or from one subgroup to another becomes an excessive burden.

My solution to this problem is to formulate all definitions relative to some carrier set of interest in such a way that the behaviour of operators or properties outside the carrier set is irrelevant. I advocate this approach in general for dealing with algebraic structures. The apparent extra complication actually achieves an economy, because when one is working with substructures, the operators and properties can all be those of the containing structure: you have no need to restrict them to the substructures or to worry about coercing the operations of one substructure into the operations of another.

As an example, I take the operations on a group G to be total functions on the universe of the type of its elements whose behaviour outside the carrier set of G is immaterial. The operations on a subgroup H of G must be represented by the same total functions. This involves no loss of generality and removes a much complexity in specifications and proofs. It may be objected that this approach gives the wrong notion of equality for groups (since the same group can be represented using two different ways of totalising the operations). However, in normal algebraic practice, one almost never needs to assert equality (as opposed to isomorphism) between two groups that are not known to be subgroups of some other group, and in that case equality has the usual meaning.

Using this approach, the three isomorphism theorems and the Cayley representation theorem are very easy to prove once one has derived the usual laws of equational reasoning in a group from the defining properties (and developed proof procedures to automate the application of these laws). Once the formalisation details were settled, it was routine and quick to prove these results.

In fact, I feel that the treatment in this case study demonstrates that polymorphic simple type theory is actually more natural than set theory for carrying out much of mathematics. For example, one can give the following very convenient definition of the symmetric group on a set X (i.e., the group comprising all permutations of the set).

$$
\begin{array}{|l}
\textbf{\textit{SymGroup}} : {'a}\ SET \rightarrow ({'a} \rightarrow {'a})\ GROUP \\
\hline
\forall\ X \bullet\ SymGroup\ X = (\\
\quad \{f \mid OneOne\ f \wedge Onto\ f \wedge \forall y \bullet \neg y \in X \Rightarrow f\ y = y\},\ (* \ Carrier\ set\ *) \\
\quad (\lambda f\ g \bullet \lambda x \bullet f(g\ x)),\ (* \ multiplication\ *) \\
\quad (\lambda x \bullet\ x),\ (* \ unit\ \ element\ *) \\
\quad Inverse\ (* \ inverse\ *) \\
)
\end{array}
$$

Here the quadruple giving the structure has components as indicated by the comments and *Inverse* is the function that maps a 1-1 onto function to its inverse function. This definition has numerous advantages over the untyped set-theoretic version. In particular, if X is a subset of Y, then the symmetric group on X is a subgroup of the symmetric group of Y as it stands, whereas this is only true "up to an isomorphism" in the standard set-theoretic account. Moreover, we can think of *SymGroup* $\{x \mid x = x\}$ as denoting the group of all permutations of the universe, sitting naturally inside the monoid of all self-mappings of the universe. This works very pleasantly: as the Cayley representation theorem states, any group is isomorphic to a group of permutations and so composition of 1-1 onto functions provides a universal prototype for the multiplication in a group, a fact which cannot even be stated properly in first-order set theory.

Moral 5: *Pace* Quine [11, article on "Mathematosis"], in a typed theory it is counter-productive to define the concept of a group so that the carrier set can be recovered from the set that represents the multiplication.

3.3 Topology

The case study in topology is perhaps the most advanced of the three in educational terms, but it still really only prvoides the beginnings of the subjects it deals with. The subjects covered are:

- abstract topology: topologies; construction of new topologies from old as (binary) product spaces or subspaces; continuity, Hausdorff spaces; connectedness; compactness.
- metric spaces: the definitions of metrics and product metrics and the result that product metrics induce product topologies; existence of Lebesgue numbers for open coverings of compact metric spaces.
- topology of the line and the plane: characterisation of connected subspaces of the line; continuity of addition and multiplication as functions on the plane.
- elementary homotopy theory: definitions of path-connectedness, the homotopy relation and the fundamental groupoid; proof that the homotopy relation is an equivalence relation and that the fundamental groupoid is a groupoid[8]

The definition of a topology is the usual one: a topology is a family of sets (referred to as *open sets*) that is closed under arbitrary unions and binary intersections.

[8] In fact, at the time of writing, all the theorems needed to justify the construction of the fundamental groupoid as a quotient of the path space have been proved, but these have not yet been brought in line with the theory of equivalence relations in [2].

$$\textbf{\textit{Topology}} : {'a}\ SET\ SET\ SET$$

$$Topology =$$
$$\{\tau \mid (\forall V \bullet\ V \subseteq \tau \Rightarrow \bigcup\ V \in \tau) \wedge (\forall A\ B \bullet A \in \tau \wedge B \in \tau \Rightarrow A \cap B \in \tau)\}$$

Since the carrier set of a topological space can readily be recovered as the union of all its open sets, the complications with signatures that arise in algebra do not arise. The signature has but one component and the thinking we decried for algebraic structures in Moral 3 now turns out to be very convenient.

The central notion of continuity takes the following form (defining an operator *Continuous* which is written postfix). Here σ and τ are intended to be topologies (and will be in the statements of all theorems that use this definition). As in the group theory case study, we work throughout with ordinary HOL total functions, taking care to make the definitions of concepts such as continuity ignore the behaviour of the functions outside some carrier set of interest, in this case the *Space* of the topologies, defined as the union of their open sets as discussed above.

$$\textbf{\$\textit{Continuous}} : ({'a}\ SET\ SET \times {'b}\ SET\ SET) \rightarrow ({'a} \rightarrow {'b})\ SET$$

$$\forall \sigma\ \tau \bullet \quad (\sigma, \tau)\ Continuous =$$
$$\{f$$
$$\mid \quad (\forall x \bullet\ x \in Space\ \sigma \Rightarrow f\ x \in Space\ \tau)$$
$$\wedge \quad (\forall A \bullet\ A \in \tau \Rightarrow \{x \mid x \in Space\ \sigma \wedge f\ x \in A\} \in \sigma)\}$$

Again, as in the group theory, this approach has the merit of localising complexity in the definitions which would otherwise spread to other definitions and to the statements and proofs of theorems. If you try to mimic the representation of functions in set theory, functions have constantly to be restricted to subspaces, whereas this is unnecessary with the total function approach.

Space does not allow an extended discussion of the methods of proof in this case study. However, there is one open problem that is worth mentioning. There is a constant need in topological reasoning to prove that functions are continuous. In algebraic topology, functions are often constructed by patching together functions defined on subspaces of the domain. For example, in proving that addition of paths in the fundamental groupoid is associative, the following result is needed, where $Open_R$ denotes the usual topology on the real line.

$$\forall k \bullet \quad (\forall t \bullet \quad k\ t =$$
$$\quad\quad\quad\quad if\ t \leq 1/4\ then\ \mathbb{NR}\ 2{*}t$$
$$\quad\quad\quad\quad else\ if\ t \leq 1/2\ then\ t + 1/4$$
$$\quad\quad\quad\quad else\ (1/2){*}t + 1/2)$$
$$\quad \Rightarrow \quad\quad k \in (Open_R, Open_R)\ Continuous$$

The proofs of such facts are very mechanical and are reminiscent of what the automated proof procedures for continuity of algebraic combinations of continuous functions in the analysis case study do. However, there are two slight complications: *(i)*, you need to apply a simple "patching" lemma to justify the continuity of a function defined by cases and *(ii)*, in the general case some small amount of intelligence is needed to pick the right topologies on intermediate sets. For example, to show that a composite $f \circ g$ is continuous with respect to a topology σ on the domain of g and a topology τ on the range of f, you need to pick some topology on the range of g that makes both f and g continuous. An algorithm to automate these proofs would be a great boon, but I do not yet have one. Joe Hurd's work on predicate subtyping [9] looks like a promising source of ideas.

Moral 6: There are a lot of new and challenging problems for proof automation in pure mathematics.

4 Will it scale?

An important question to ask of any case study in applying formal methods and theorem-proving in engineering applications is "will the proposed technique scale to real-life applications?". I believe the same applies to mathematical applications as well. Simpson [12] identifies what is probably one of the most important problems for more advanced pure mathematics: much use is made of structure which share a combination of algebraic, topological or geometrical properties. For example, the rich and important theory of Lie groups is an abstraction of the algebraic and geometric theory of groups of real or complex matrices. A Lie group is simultaneously a group and a smooth manifold, a smooth manifold being something that has a particular topological structure combined with a differential structure allowing analytic methods to be used. The issue then is, how to deal formally with the kind of reasoning that is endemic in mathematics where one just says something like "let G be a Lie group" and then freely appeals to the notations and theory of whichever of the underlying structures provides the facts one needs.

I believe the approach to modelling mathematical structures exemplified by the case study on groups and also by the $\mathbb{Z}$-set example in section 2 above will scale, subject to some slight modifications to the details, ideally supported by some extensions to the syntax offered by the parser (see [2] for more details on the latter).

The main change to the approach addresses the issue highlighted by Simpson in his example of Lie groups. To get things to scale, I would propose using labelled products rather than unlabelled products for the signatures of algebraic structures. To see how this would work, consider the notion of a field: Given

our treatment of groups, a field can conveniently be thought of as two group structures on elements of the same type obeying certain laws[9].

Using labelled products, the signature for groups would be given by the following ProofPower-HOL labelled product type definition which defines a new polymorphic labelled product type *'a GROUP* with four components with the indicated labels and types. The component labels become the names of the functions that project the product type onto its component types.

HOL Labelled Product

GROUP

Car_G	$: 'a\ SET$;
$Times_G$	$: 'a \rightarrow 'a \rightarrow 'a$;
$Unit_G$	$: 'a$;
$Inverse_G$	$: 'a \rightarrow 'a$

Now we can define the signature for a field as a labelled product. Note that in both these labelled product definitions, in the interests of scalability to complex situations, we are decorating the component labels with subscripts to avoid clashes with other algebraic structures, e.g., rings would also have an additive group.

HOL Labelled Product

FIELD

$AdditiveGroup_F$	$: 'a\ GROUP$;
$MultiplicativeGroup_F$	$: 'a\ GROUP$

This captures the desired semantics, but creates some syntactic problems. For example, the expression 1 + x.y in a field K would have to be written.

$$Times_G\ (AdditiveGroup_F\ K)\ (Unit_G(MultiplicativeGroup_F\ K)\ )\ (Times_G(MultiplicativeGroup_F\ K)\ x\ y)$$

This syntactic problem can be overcome either by explicitly defining accessor functions as we did for $\mathbb{Z}$-sets and groups, or by extending the parser and typechecker to allow aliases for non-constant expressions, or perhaps, specifically for composite functions, so one could define + and . to be aliases allowing something like $1 + Kx.Ky$ to be written for the above term.

Simpson proposes a solution in dependent type theory to this problem in which mathematical structures are represented by functions from strings to component structures. This is not available to us in HOL, but I can think of no examples in mathematics where the statically typed approach sketched above would be semantically insufficient.

[9] This does not work in the traditional set-theoretic account, since the multiplicative structure of a field does not comprise a group unless it is restricted to the non-zero elements.

5 Summary

I have given an overview of three case studies in the use of the ProofPower-HOL theorem-prover on pure mathematical problem domains. This has highlighted some problems in giving a smooth formalisation. Solutions or partial solutions to these problems have been proposed. In particular, I have outlined a method for scaling the approach to the compound mathematical structures that predominate in modern century mathematics.

I have extracted some "morals" from the work done to date, and there is much more that could be said about good ways to go about capturing a useful and evolving body of pure mathematics in an automated theorem-proving system. However, agonising about the technical approach will be less productive than actually trying to do some mathematics and to learn from the attempt.

Acknowledgments

My thanks are due to Matthew Franks, Hanne Gottliebsen, John Harrison, Roger Jones and Larry Paulson for their very helpful correspondence during the development of these case studies.

References

1. R.D. Arthan. Mathematical case studies: Basic analysis. `http://www.lemma-one.com/ProofPower/examples/examples.html`, 2004.
2. R.D. Arthan. Mathematical case studies: Some group theory. `http://www.lemma-one.com/ProofPower/examples/examples.html`, 2004.
3. R.D. Arthan. Mathematical case studies: Some topology. `http://www.lemma-one.com/ProofPower/examples/examples.html`, 2004.
4. Hanne Gottliebsen. *Automated Theorem Proving for Mathematics: Real Analysis in PVS*. PhD thesis, University of St. Andrews, 2001.
5. T. Hales. The Flyspeck Project Fact Sheet. Technical report, `http://www.math.pitt.edu/~thales/flyspeck/index.html`, 2003.
6. John Harrison. Formalized Mathematics. Technical report, `http://www.cl.cam.ac.uk/users/jrh/papers/`, 1996.
7. John Harrison. Theorem Proving with the Real Numbers. Technical report, University of Cambridge Computer Laboratory, 1996.
8. Michael Henle. *A Combinatorial Introduction to Topology*. Dover Publications, Inc., 1979.
9. Joe Hurd. Predicate Subtyping with Predicate Sets. In Richard J.Boulton and Paul B. Jackson, editors, *Proceedings of TPHOLs 2001, LNCS 2152*. Springer-Verlag, 2001.
10. L. Paulson. Defining functions on equivalence classes. *Preprint: available at* `http://www.cl.cam.ac.uk/users/lcp/papers/Reports/equivclasses.pdf`, 2004.
11. W.V. Quine. *Quiddities*. Harvard University Press, 1987.
12. Carlos Simpson. Computer Theorem Proving in Math. *arXiv:math.HO/0311260 v2*, 20 February 2004.
13. J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992.

A THE THEORY $\mathbb{Z}$_set

A.1 Parents

$\mathbb{Z}$

A.2 Constants

$**	$'a \to \mathbb{Z} \to 'a\ SET \times (\mathbb{Z} \to 'a \to 'a) \to 'a$
Car	$'a\ SET \times (\mathbb{Z} \to 'a \to 'a) \to 'a\ SET$
$\mathbb{Z}$_***Set***	$('a\ SET \times (\mathbb{Z} \to 'a \to 'a))\ SET$
Orbit	$'a\ SET \times (\mathbb{Z} \to 'a \to 'a) \to 'a \to 'a\ SET$

A.3 Type Abbreviations

$'a$ $\mathbb{Z}$_***SET***	$'a\ SET \times (\mathbb{Z} \to 'a \to 'a)$

A.4 Fixity

Infix 310: **

A.5 Definitions

Car

**

$$\vdash \forall\ (set,\ action) \bullet\ Car\ (set,\ action) = set \wedge (\forall\ x\ i \bullet\ (x\ **\ i)\ (set,\ action) = action\ i\ x)$$

$\mathbb{Z}$_***Set***

$$\vdash \forall\ X \bullet\ X \in \mathbb{Z}\_Set \Leftrightarrow (\forall\ x\ i \bullet\ x \in Car\ X \Rightarrow (x\ **\ i)\ X \in Car\ X) \wedge (\forall\ x\ i\ j \bullet\ x \in Car\ X \Rightarrow (x\ **\ (i + j))\ X = ((x\ **\ i)\ X\ **\ j)\ X) \wedge (\forall\ x \bullet\ x \in Car\ X \Rightarrow (x\ **\ \mathbb{NZ}\ 0)\ X = x)$$

Orbit

$$\vdash \forall\ X\ x \bullet\ Orbit\ X\ x = \{y | \exists\ i \bullet\ y = (x\ **\ i)\ X\}$$

A.6 Theorems

$\mathbb{Z}$_set_minus_thm

$$\begin{array}{l}\vdash \forall\ X \\ \quad \bullet\ X \in \mathbb{Z}\_Set \\ \qquad \Rightarrow (\forall\ x\ i \bullet\ x \in Car\ X \Rightarrow ((x\ **\ i)\ X\ **\ \sim i)\ X = x)\end{array}$$

$\mathbb{Z}$_set_cancel_thm

$$\begin{array}{l}\vdash \forall\ X\ x\ y\ i \\ \quad \bullet\ X \in \mathbb{Z}\_Set \wedge x \in Car\ X \wedge y \in Car\ X \\ \qquad \Rightarrow ((x\ **\ i)\ X = (y\ **\ i)\ X \Leftrightarrow x = y)\end{array}$$

orbit_refl_thm

$$\vdash \forall\ X \bullet\ X \in \mathbb{Z}\_Set \Rightarrow (\forall\ x \bullet\ x \in Car\ X \Rightarrow x \in Orbit\ X\ x)$$

orbit_disjoint_thm

$$\begin{array}{l}\vdash \forall\ X \\ \quad \bullet\ X \in \mathbb{Z}\_Set \\ \qquad \Rightarrow (\forall\ x\ y \\ \qquad \bullet\ x \in Car\ X \wedge y \in Car\ X \\ \qquad\quad \Rightarrow Orbit\ X\ x \cap Orbit\ X\ y = \{\} \\ \qquad\qquad \vee\ Orbit\ X\ x = Orbit\ X\ y)\end{array}$$

A Framework for Interactive Sharing and Deductive Searching in Distributed Heterogeneous Collections of Formalized Mathematics

James L. Caldwell[1] * and Christoph Jechlitschek[2] **

[1] Department of Computer Science, University of Wyoming, Laramie, WY
[2] Department of Computer Science, Washington University, St. Louis, MO.

Abstract. Peer-to-peer technology implemented in systems like Napster allowed sharing of digitized music across the web in an incredibly easy to use system. This paper describes a prototype peer-to-peer system for networking distributed and heterogeneous databases of formalized mathematics. We also propose a general framework for deductive search in heterogeneous libraries of formal content. As participants in this conference well know, a significant body of mathematics has been formalized in theorem provers. We believe that a truly distributed mechanism for sharing formal content will multiply efforts of individual users of theorem proving systems, will invigorate ongoing formalization efforts, and will spur new research in deductive search and content-based addressing. Interactive sharing has the potential to be a significant new methodology for theorem proving. A basic tenet of our approach is that users of the system must be able to account for results and methods for accountability are incorporated into the proposed methods.

1 Introduction

We imagine a future in which the web plays an integral role in theorem proving efforts. Where theorems and proofs of diverse systems are interactively searched by developers across the web and where sharing is used to discharge significant numbers of proof obligations.

There is a diverse array of theorem proving systems representing many hundreds of man-years of effort, they range from those that completely automate the proof search to interactive proof checkers. A list of these systems would includes ACL2, Coq, Elf, HOL, Isabelle, MetaPrl, Mizar, Nuprl, PVS and others. The number of extant theorems that have been proved in these systems is astounding. (The reader is invited to make his own estimate). Currently, the costs

* The first authors work was partially supported by NSF grant CCR-9985239.
** This work was performed at the University of Wyoming and was supported by a DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under grant N00014-01-1-0765.

of sharing formal material are so high that little sharing takes place, sometimes even within communities of users of the same tool. Much of the sharing that does take place requires personal communication between the parties involved.

In this paper we describe a vision of the future and argue that it need not merely be a fantasy. Toward this end we describe the implementation of a prototype peer-to-peer framework for connecting distributed libraries of mathematics [17]. The prototype implementation also supports a query language for content and name based searching. We go further in proposing a general framework for deductive search methods in a collection of logically heterogeneous databases[3]. Our approach is guided by our experiences in the Formal Digital Libraries project [8], a joint project between Cornell, Caltech and Wyoming.

1.1 Vision

We imagine a system in which proof obligations may be discharged by existing results, that have perhaps been verified in different logics, and recorded in a distributed and heterogeneous database. Consider the following scenario.

> A user, sitting in Laramie Wyoming on a blustery winter evening, is constructing a proof in the Nuprl system[4]. At various points in the process, suspecting that surely someone else has already proved the result required to complete her proof, she initiates an interactive query. Moments before the Wyoming user issued her query, an early rising HOL user in the warm summer morning of Canberra Australia has just proved a lemma having the required semantic import. Upon completing his proof it was automatically committed to his local online database. Our Wyoming users query returns the HOL result together with a procedure for translating HOL terms into Nuprl terms and includes evidence that the proof actually was completed in HOL. This information is incorporated into her local database and used to complete her proof. Once completed, her new result is recorded into her data-base thereby making it immediately available to other proof efforts distributed across the web.

In this paper we argue that this scenario is both theoretically feasible and practically realizable with existing technologies (circa 2004). We present evidence for this argument by describing a prototype implementation of a peer-to-peer network for interconnecting databases of formal mathematical content. We continue by outlining a general theoretical framework for deductive searching in distributed networks libraries of heterogeneous formalized mathematics.

We reckon that the following are the necessary components for such a system.

[3] Throughout the paper, the words "library" and "database" should be considered synonymous, though perhaps the word "database" emphasizes implementation.

[4] By inclination, our hypothetical user is interested in extracting programs from proofs but has no philosophical objections to incorporating classical results into her proofs if it does not impinge on the constructive content. For a discussion of just such a methodology for incorporating classical results into constructive proofs see [5].

i.) Individual databases of formalized mathematics.
ii.) A framework for connecting individual databases into a distributed network including methods for finding databases of formal material and a protocol for communicating between them.
iii.) Methods of translating between logical theories.
iv.) Methods of searching across the distributed network.

There are independent research efforts underway on all these topics. What does not currently exist is an effort to pull these technologies together into an unified approach. In this paper we address items (ii), (iii) and (iv). We do not propose to constrain (i) other than to require that databases participating in the network implement the protocol described as part of (ii). We believe that a successful implementation of items (i), (ii) and (iv) will create a *market* that will further stimulate the development of translations (iii).

We remark that, perhaps surprisingly, scientific communities other than computer science seem to be better at sharing results in a significant way; by which we mean that information is shared so that it can be used directly in establishing new results, not simply in a secondary form. For example, the databases of genetic structures are remotely accessible and remote access forms a crucial part of the methodology used by researchers working in that field. On the homepage for the National Center for Biotechnology Information [22] it says: "Most journals now expect that DNA and amino acid sequences that appear in articles will be submitted to a sequence database before publication." As a community, we could take a lesson here.

1.2 Relating theories

The ability to soundly combine theorems proved in different logics within the same framework is a deep mathematical problem. Institutions [10, 11] provide a category theoretic framework in which the formal relations between different theories can be established. Although institutions provide a mathematical framework within which relations between logics can be understood, they have been little used in practice. The hard part of relating theories is establishing the semantic map. Howe [15, 14] has provided the semantic foundations for a map between HOL and a classical variant of Nuprl. An implementation of the translation is described in [21]. Naumov [20] has related Isabelle and classical Nuprl and a semantic justification for translating PVS results into classical Nuprl has recently been completed [19]. Staples has related ACL2 and HOL [25] providing a mechanism to incorporate results of ACL2 into HOL proofs. Applications for sharing results (even the use of classical results from PVS in constructive Nuprl proofs) are discussed in [5]. In each case, a semantically justified translation from the language of one logic into the language of another is required.

Applying an economic model, we note that translations between theories are implemented by individuals who value the incorporation of results from one theory into their own highly enough to do the required work. Part of the calculation of the worth of such an effort is based on the amount of and type of material that

will be made accessible by a translation and the ease with which it will be used by the developer and others. The framework proposed here lessens the effort required to apply such translations, *i.e.* it provides a market for such tools. By providing a such a market, we believe that a framework such as the one described here for integrating multiple provers will motivate further developments.

1.3 Accountability

A guiding principle of our framework and of the Formal Digital Library [8] (FDL) is one of accountability. Consumers of theorems and other formal objects have a right to know what assumptions, facts and methods an object depends on; this problem has seen previous study [12]. Only with this knowledge can users make epistemic judgments whether to accept results and to incorporate them into their own work. As part of the FDL effort, Allen [2] has designed a novel mechanism to certify facts about objects in a database of terms. These certifications carry epistemic weight in that: users may create new *certificate kinds*, they may request than an existing kind of certification be run on a particular object, or they may examine existing certificates. Users may not create certification objects, only the system can do so by evaluating the computational part of a certificate kind. Users can determine exactly what has been certified by examining the code used to create a certificate. In the scheme of the FDL, there are a plethora of certificates generated by many users; some may be as simple as a claim that some individual created the certified object, others may certify that a proof has been accepted by some formal tool or that some object originated from a particular database. This certification mechanism can be used to build sets of dependencies and properties of objects and to track them. Users can inspect certificates and, by evaluating the methods used to generate a particular certificate kind, can determine the epistemic weight they accord to the certified object.

Accounting for the correctness of a formal object (let's say a proof) depends on a complex set of facts that at least include which tools (and version) were used to produce the proof; the lemmas, tactics, and methods of proof the theorem itself depends on; global settings in the environment when the proof was done; and perhaps other facts. This list must be open-ended since the evidence required for an individual to accept a result ultimately depends on that individuals personal criteria. The criteria for believing something can vary from individual to individual and thus, the threshold of evidence may be higher or lower, depending on the individual. In an extreme case, users may accept results based on authority *e.g.* *'Caldwell said "Constable said ϕ is a theorem."'* But even this form of evidence[5] may carry epistemic weight with users and our goal is to include even this kind of evidence. Every kind of formal object potentially requires some form of evidence (formal or informal) to justify its use in certain contexts.

[5] Evidence like this may actually be easy to account for using certificates based on digital signatures.

1.4 Searching

We intend to search in heterogeneous databases, *i.e.* databases containing results from a number of logical systems. The effectiveness of existing search technologies would seem to be the principal technical obstacle to true integration of these ideas into proof engines.

There are two aspects to the search problem. The first is to find the available databases of formal content on the web that are open to pubic search; the second is to search those databases for formal content (definitions, theorems, proofs, translations, tactics, *etc.*) in a semantically robust way. The first problem is addressed (and solved) by our prototype peer-to-peer network. The second problem is theoretically challenging and open ended in that we expect new search methods will constantly be developed. Below we describe a framework for deductive search within which we believe new methods can be couched.

Formalized mathematical proofs and theories are fragile objects[6] and although the semantic import of a theorem may well match or subsume a lemma being searched for, the shape of the theorem may not trivially match the search pattern. Trivial syntactic differences in theorems having little or no semantic content (*e.g.* $\forall x.\phi \wedge \psi$ vs. $\forall y.\psi \wedge \phi$) can make naively implemented search fail, users would be disappointed with such failures. Also, the equivalence of formulas in different logics differ, *e.g.* classically, $\phi \Rightarrow \psi$ and $\neg\phi \vee \psi$ are equivalent while they are not equivalent in the constructive setting; this must be taken into account in a heterogeneous setting by specifying the logic to use for deduction in search.

Methods for searching formal content might be based on unification[7] [7], but other strategies are possible as well. Of course, the problem of determining if a previously verified lemma (or collection of lemmas) subsumes a query target is undecidable in general.

2 A Peer-to-peer framework

The second author has built a prototype peer-to-peer network for sharing information between FDL's. The details of the architecture and of the implementation are described in [17]. Figure 1 gives an overview of the architecture. Peer-to-peer applications are becoming ubiquitous; they are used to share files, CPU cycles, and other resources. A principal advantage of the technology is its inherent fault-tolerance, there is no centralized component to fail. Peer-to-peer networks also

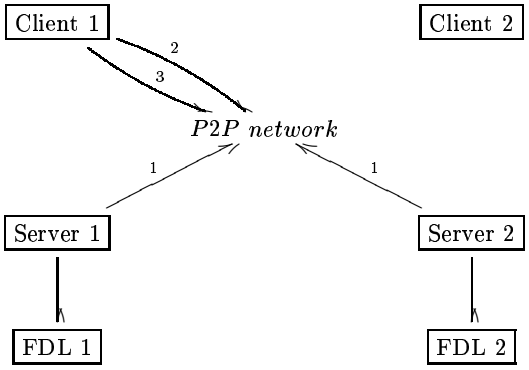
[6] Even tyros have first hand experience of this fragility. Small changes, *e.g.* adding an antecedent to the statement of a putative theorem, will often break a partial derivation that may have already been constructed. In fact, the most experienced users of such tools distinguish themselves from novices in that they build proofs in such a way as to avoid failure under minor perturbations to the statement being verified.

[7] Higher-order unification is undecidable but unification based methods can still be used since a user only needs a non-empty approximation to the complete search to satisfy a query.

support distributed discovery mechanisms. Sun has developed an open source peer-to-peer framework called JXTA [18] that is platform and programming language independent. Our system is built on JXTA.

2.1 A Prototype Implementation

The prototype is implemented in Java. It consists of about 6000 lines of code and includes a name and content based search engine for FDLs. The JXTA framework is used to provide the peer-to-peer network functionality.



Server 1 and Server 2 advertise the existence of the libraries FDL 1 and FDL 2 to the P2P network (step1). An interested client discovers these libraries (step 2). The client then queries the servers over the P2P network (step 3).

Fig. 1. P2P architecture

The FDL provides a TCP/IP based mechanism that allows clients to connect to the library and to issue simple search requests. The current interface to the library is limited to a search by name request and a search by content request. The search by name request returns a set of all theorems that contain a given string as a substring of their names. The search by content request returns the set of all theorems whose statements contain operators specified in the search. We developed a Boolean query language using the logical operations 'and', 'or', and 'not' to create more powerful expressions. While the query language is very simple it is surprisingly useful and serves to prove the mechanisms for searching remote libraries work. Indeed we found many new theorems in the Cornell libraries while testing our tool. Within the prototype, the search engine is implemented modularly and can easily be replaced if extensions are required.

To share theorems between groups we not only need to be able to search libraries but we also we need to discover the libraries themselves. In peer-to-peer networks, servers and clients have equal rights and responsibilities and are

connected in a mesh topology. Peer-to-peer networks support mechanisms for discovering other peers and exchanging information between them. The JXTA framework provides a high level abstraction of these mechanisms. In our prototype, the libraries offering search services advertise it in the network. Clients can use those advertisements to invoke the services. All communication between peers is done within the peer-to-peer network having the advantage that problems with firewalls can be avoided (see [17] for details.) Since databases are not designed to participate directly in the peer-to-peer network a small server application was developed which is deployed in front of each library. Not only does this provide the interface for application libraries to join the peer-to-peer network, the server could also be used to provide additional functionality like authentication, authorization, and accounting.

3 A Framework for Deductive Search

In this section we describe the framework within which we address the problem of searching in distributed heterogeneous databases of formalized mathematics. The proposed framework is intended to be independent of the underlying individual databases; although we have in mind the FDL. The proposed framework does not make assumptions about the underlying databases but assumes that they provide a uniform interface; we (partially) specify that interface here.

The framework consists of the following components.

i.) A term structure used to communicate information across the network, the class of terms is denoted $Term_I$.[8]

ii.) An application programmers interface (API) supported by databases in the network.

iii.) A peer-to-peer architecture for the interconnection of the databases supporting functions for dynamically integrating new databases into the network and the protocol for communication between them.

iv.) A logical framework imposed on terms for describing the methods of deductive search. This is based on a concept of formal languages as decidable subclasses of terms in $Term_I$. These languages include the languages of the various logics together with the other extra-logical languages; *e.g.* representations of executable code (*e.g.* tactics, translations and others) together with all the other components necessary for the representation and manipulation of formalized mathematics. We also intend that informal content will be representable in the database as well.

The communication between systems is facilitated by a uniform and extensible term structure. This is the same term structure used internally by the FDL to represent formalized mathematics though we do not assume it is the internal representation used by all databases connected in the network; simply that they can translated their internal representations into the specified form. We also use the term structure in the description of the framework for deductive searching.

[8] The term structure described here is based on Nuprl's term structure [1] and is the one used internally within the FDL; we use it as an interface language.

3.1 Terms and Languages

The issues related to the representation of formalized mathematics are extraordinarily complex, especially as related to binding structures[9]. In this section we present the term structure used in the FDL which offers some generality in binding.

$Term_I$ is the class of recursive tree structures of the form

$$opid\{parameters\}(bterms)$$

where *opid* is the operator name, *parameters* is a list of value-type pairs and *bterms* is a list pairs consisting of a list of variables and a term. The parameter I is the class of abstract atomic identifiers allowing terms to refer to other terms in $Term_I$.

$$\begin{array}{ll}
\langle Term_I \rangle & ::= D \mid \langle opid \rangle \{ \langle parameter \rangle^* \} (\langle bterm_I \rangle^*) \\
\langle opid \rangle & ::= \langle C \rangle \langle C \rangle^* \\
\langle bterm_I \rangle & ::= \langle vars \rangle^* . \langle term_I \rangle \\
\langle parameter \rangle & ::= \langle value \rangle, \langle type \rangle \\
\langle C \rangle & ::= any\ character
\end{array}$$

Parameters are constants or other arguments not constituent in the subterms *e.g.* within the FDL representation of Nuprl and PVS terms, the number 1 is a constant term of the form `natural{1:num}()`, the string "xyzzy" is represented by the term `string{"xyzzy":string}()`. The class of parameters is not fixed and can be extended to accommodate new languages and logics.

The *bterms* are the subterms of a term, possibly containing bound variables. A bterm consists of a list of variables (the bound variables) and a term (the body). Occurrences of variables included in the list of bound variables are bound in the body of the bterm. The use of bterms to encode binding operators can be seen by considering the encoding of the lambda abstraction in this structure. The term $\lambda x.M$ is encoded as `lambda{}(x.`M`)`. The opid of this term is `lambda`, it has no parameters, and it has one subterm M in which occurrences of the variable `x` are bound. The universal quantifier $\forall x{:}T.P$ is encoded as `all{}(`T`;x.`P`)`. The operator id is `all`, there are no parameters, and the operator has two subterms, T (the domain from which the bound variable `x` is chosen, and the bound term `x.`P where P is a term possibly containing free occurrences of the variable `x`. The fact that there may be multiple variables bound simultaneously in a subterm allows for the specification of an operator like Nuprl's *spread* operator, a generalized destructor for pairs; it is defined as `spread{}(`p`;`$x,y.t$`)`. The computation rule for spread makes clear how the simultaneous binding is used when the subterm p is a pair.

$$\texttt{spread(<}a,b\texttt{>; }x,y.t\texttt{)} \rightarrow t[a,b/x,y]$$

[9] For an interesting discussion of alternative binding structures see [13] and references therein.

i.e. if the first argument to `spread` is a pair of the form $\langle a, b \rangle$, spread simultaneously substitutes the first element of the pair for x in t and the the second element of the pair for y in t.

The index set I is not necessary for representing individual terms of a logic. By providing a means for terms to refer to other terms, the identifiers in the set I allow arbitrarily complex structures to be embedded within a collection of terms. Formal libraries are such structured collections. Allen has argued in detail elsewhere [2] that the references between terms should be abstract and atomic, thus the identifiers in I have no discernible structure and simply serve to refer. Indeed, within the conception of the FDL, the only significant property of the indices in a structured collection of terms is the topology of the constituent components induced by the references between the terms. More precisely, if I is the set of abstract identifiers used in $Term_I$ and I' is a set of abstract identifiers of equal or greater cardinality, then the process of uniformly replacing the abstract identifiers in a database of terms in $Term_I$ (under any injective map from I to I') results in a database of terms in $Term_{I'}$ which carries the same semantic import as the original.

This term structure has been used to represent both Nuprl, HOL and PVS terms in the FDL [8, 3].

A *language* is a decidable subset of terms *i.e.* $\mathcal{L}$ is a language if $\mathcal{L} \subseteq Term_I$ and for every $t \in Term_I$, we can decide if $t \in \mathcal{L}$. We assume interesting languages have names and abuse our own notion by identifying $\mathcal{L}$ both with the set of terms in the language and as a name of the set of terms. If $\mathcal{L}$ is a language we also use the name $\mathcal{L}$ to denote the property of membership in $\mathcal{L}$, thus if $\mathcal{L}$ occurs as a property it denotes the property $(\lambda t.\, t \in \mathcal{L})$. We note here that many of the languages we are interested in will be the terms of some logic, though not all interesting languages are logical.

3.2 Databases and Filters

In our model, libraries are collections of terms that refer to one another via the abstract atomic identifiers together with collections of certificates making claims about the stored terms.

Every individually stored term has an index $i \in I$ and terms may contain indexes to other terms. There is no requirement that every subterm of a term be indexed, though it is possible to build terms by storing subterms individually and referring to them by their abstract identifiers.

The evidence associated with a term is carried in the certificates for the term. We use the Greek 'ϵ', possibly decorated, as meta-variables denoting evidence. Terms retrieved from databases are packaged with the evidence associated with them and we call such packages *eterms*. We denote the type of evidentiary terms $Term_{I\mathcal{E}}$ and write t_ϵ to denote elements of this type. Evidence can be erased from an evidentiary term, $\lceil t_\epsilon \rceil = t$, *i.e.* $\lceil \cdot \rceil : Term_{I\mathcal{E}} \to Term_I$ and similarly, evidence can be garnered from an eterm $\lfloor t_\epsilon \rfloor = \epsilon$. No mechanism is provided for evidentiary terms to be composed (except by the database itself) and we expect

to apply encryption mechanisms to enforce the constraint that only the database can deliver an eterm.

Once exported from a database, every term has at least one piece of evidence associated with it which an identifier of the database it originated in. Of particular practical interest and current research is the problem of how evidence in the form of certificates can be transferred from one database to another without forcing the re-verification of the certificates. We expect that methods based on digital signatures, like that described in [12], can be applied to this problem.

Within a database, term indexes (either stored as data or computed as needed) are used to select objects satisfying some property *e.g.* the terms of the PVS logic, or the Nuprl tactics. Term indexing is a tool to pair down the search space before the computationally expensive part of the search is performed by filtering objects that are unlikely to match. See [24] for efficient data-structures and algorithms for term indexing of first-order terms. We imagine many such indexing operations will be defined and provide the framework for specifying them here.

Given a database $\mathcal{D}$ of terms and a property ($\varphi : Term_{I\mathcal{E}} \to \mathbb{B}$) of terms, $\mathcal{D}{\downarrow}\varphi$ is the set of eterms in $\mathcal{D}$ satisfying φ:

$$\mathcal{D} \downarrow \varphi \stackrel{def}{=} \{ t_\epsilon \in \mathcal{D} \mid \varphi[t_\epsilon] \}$$

If $P = \{\varphi_1, \cdots, \varphi_n\}$ is a set of properties of eterms, we write $\mathcal{D} \downarrow P$ to denote the set of terms satisfying at least one of the properties in P *i.e.* $\{\varphi_1, \cdots, \varphi_n\}$ is a notation for the property $(\lambda t_\epsilon . \varphi_1[t_\epsilon] \vee \cdots \vee \varphi_n[t_\epsilon])$. Note that the fact that properties are defined on eterms means we can filter databases by syntactic properties of the terms and/or by the evidence the terms carry.

If $\mathbf{D}$ is any set of databases and if φ is any property of terms, then:

$$\mathbf{D} \downarrow \varphi \stackrel{def}{=} \bigcup_{\mathcal{D} \in \mathbf{D}} \mathcal{D} \downarrow \varphi$$

Individual databases may vary in their underlying implementations though they must all support translations from their internal representations into the term representation that serves as the medium of communication between systems. A framework like the one proposed here, characterized by operations on terms, allows for specification of search methods in terms of the interface language.

3.3 Translations

Our methodology for sharing results rests on the idea that there may be effective translations between logics. In [26], an application similar to the one here is given which accounts for the use of multiple logics within a single specification.

If there is a partial function f mapping terms to terms ($f : Term_I \to Term_I$) such that the domain of f is $\mathcal{L}'$ and the codomain of f is $\mathcal{L}$ we call $\langle f, \mathcal{L}', \mathcal{L} \rangle$

a *translation*. Note that since the domains of translations may intersect, we explicitly carry the domain and codomain with the translation[10]. If f is a function from $\mathcal{L}'$ to $\mathcal{L}$ and $t \in \mathcal{L}'$ then $f(t)$ evidently denotes the translation of $t \in \mathcal{L}'$ into a term in the language $\mathcal{L}$.

We are typically interested in translations that make some kind of guarantee, *e.g.* that some property is preserved by the translation. The evidence for guarantees are represented in certificates[11] and so, a translation which generates evidence for its own correctness must generate certificates. Only the database can generate certificates and so evidentiary translations must carry references to certificate kinds (a certificate generator) and make requests to the database to execute them. A translation certificate kind (of type $\mathcal{CK}$) takes an eterm t_ϵ whose term part is of type $\mathcal{L}'$ and returns a new eterm $t'_{\epsilon'}$ where ϵ' is the new evidence for the translated term t'. As a side effect it adds the new term $(t' = f(\lceil t_\epsilon \rceil))$ to the database and creates new certificates for t' both preserving the old certificates $\lfloor t_\epsilon \rfloor$ (noting that they belonged to the untranslated term t_ϵ) and generating a new certificate certifying that t' was indeed generated by the translation of t_ϵ. If $\mathcal{C}$ is a reference to a certificate kind we write $\mathcal{C}^*(t_\epsilon)$ to denote the result of a request for the database to apply certificate kind $\mathcal{C}$ to t_ϵ.

Thus,the type of evidentiary translations is defined to be the four-tuple:

$$\mathbf{Tr} \stackrel{def}{=} (Term_I \rightarrow Term_I) \times (Term_I\,\mathbf{Set}) \times (Term_I\,\mathbf{Set}) \times \mathcal{CK}$$

If $\tau = \langle f, \mathcal{L}, \mathcal{L}', C \rangle$ is in $\mathbf{Tr}$ then:

$$\tau(t_\epsilon) \stackrel{def}{=} C^*(t_\epsilon) \qquad dom(\tau) \stackrel{def}{=} \mathcal{L} \qquad codom(\tau) \stackrel{def}{=} \mathcal{L}'$$

We define the composition of evidentiary translations $(\tau \circ \hat{\tau})$ as follows. If $\tau = \langle f, \mathcal{L}', \hat{\mathcal{L}}, \mathcal{C} \rangle$ and $\hat{\tau} = \langle g, \hat{\mathcal{L}}, \mathcal{L}, \hat{\mathcal{C}} \rangle$ are compatible translations (*i.e.* if $codom(\tau) = dom(\hat{\tau})$) then:

$$\tau \circ \hat{\tau} = \langle f \circ g, \mathcal{L}', \mathcal{L}, \mathcal{C} \circ \hat{\mathcal{C}} \rangle$$

The notation $(f \circ g)$ denotes ordinary function composition defined as $(f \circ g)(x) = g(f(x))$.

The identity translation on a language $\mathcal{L}$ is defined as $Id_{\mathcal{L}} = \langle \lambda x.x, \mathcal{L}, \mathcal{L}, \mathcal{C}_{Id} \rangle$, where $\mathcal{C}_{Id}$ is the certificate kind that has no side effects and $\mathcal{C}^*_{Id}(t_\epsilon)$ simply returns the value t_ϵ.

In practice the syntactic translations between the formal languages may be straightforward, the hard part for nontrivial translations between logics is the justification that the translation preserves intended meanings. The justification

[10] This is consistent with formalizations of category theory [16, 6] in which each arrow has a *dom* and *codom* function and so arrows in non-trivial categories are triples. With this in mind, we see that languages are the objects of the category, translations are the arrows and composition is defined as below.

[11] Certificates justifying a translation may refer to an informal argument (a paper)or they may refer to other formal content.

that the intended meaning is preserved by a translation may be informal or formal. To the extent that a user believes the justification for a translation, he will include it (or not) in the set of translations he wants considered when calculating a set of candidates for a search.

3.4 Stratification of Languages by Translations

To consider the relationships between objects in different languages in a heterogeneous database, we stratify terms relative to a fixed language $\mathcal{L}$ by their distance from that language via some sequence of translations in a specified[12] set $\mathbf{T}$. For the purposes of search, we are ultimately interested terms that can be effectively translated from one language (logic) to another. Based on this idea, we provide the following definition of the n-closure of a translation set $\mathbf{T}$ relative to a language $\mathcal{L}$.

$$\begin{array}{ll} \mathbf{T}^0_{\mathcal{L}} & \stackrel{def}{=} \{\tau : \mathbf{Tr} \mid \tau = Id_{\mathcal{L}}\} \\ \mathbf{T}^{n+1}_{\mathcal{L}} & \stackrel{def}{=} \{\tau : \mathbf{Tr} \mid \exists \tau' \in \mathbf{T}.\ \exists \hat{\tau} \in \mathbf{T}^n_{\mathcal{L}}.\ codom(\tau') = dom(\hat{\tau}) \wedge \tau = (\tau'\ o\ \hat{\tau})\} \end{array}$$

The class $\mathbf{T}^n_{\mathcal{L}}$ consists of all translations mapping terms of languages $\mathcal{L}'$ to the language $\mathcal{L}$ by a sequence of n translations from the set $\mathbf{T}$.

We define the closure of the stratification to be the union of all the levels.

$$\mathbf{T}^*_{\mathcal{L}} \stackrel{def}{=} \bigcup_{i \in \mathbb{N}} \mathbf{T}^i_{\mathcal{L}}$$

This is the set of all terms interpretable as terms in $\mathcal{L}$ by some sequence of translations in $\mathbf{T}$.

The languages at level k in $\mathbf{T}^k_{\mathcal{L}}$ can be retrieved by projecting them from the translations in that level.

$$\|\mathbf{T}^n_{\mathcal{L}}\| \stackrel{def}{=} \pi_2(\mathbf{T}^n_{\mathcal{L}})$$

where the projection functions are lifted to sets of tuples point-wise in the natural way (*i.e.* if $S \subseteq S_1 \times \cdots \times S_n$ then $\pi_i(S) = \{x_i : S_i | \langle x_1, \cdots, x_i, \cdots, x_n \rangle \in S\}$ where $0 < i \leq n\}$).

The distance of a language $\mathcal{L}'$ from $\mathcal{L}$ under the translations set $\mathbf{T}$ is defined if and only if $\mathcal{L}' \in \|\mathbf{T}^k_{\mathcal{L}}\|$ for some k and is the minimum k such that $\mathcal{L}' \in \|\mathbf{T}^k_{\mathcal{L}}\|$.

The languages included in the closure $\mathbf{T}^*_{\mathcal{L}}$ determine the potential search space (and translations to use) to satisfy a query in the language $\mathcal{L}$.

The set of terms from a collection of databases $\mathbf{D}$ under translation set T at distance k from $\mathcal{L}$ is the set $\mathbf{D} \downarrow \|\mathbf{T}^k_{\mathcal{L}}\|$. We call this set the *k-step candidate terms*. The *full set of candidate terms* are the terms in $\mathbf{D} \downarrow \|\mathbf{T}^*_{\mathcal{L}}\|$. These sets are sets of terms in the languages $\mathcal{L}'$, that can be translated into terms in $\mathcal{L}$. We are of course not only interested in the sets of terms which *can* be translated into the language $\mathcal{L}$ but are interested in their translations. The *effective candidate*

[12] We specify the set of allowable translations $\mathbf{T}$ because it is a basic tenet of our approach that users must be able to account for the results they receive.

terms of $\mathcal{L}$ from $\mathbf{D}$ under $\mathbf{T}$ is the set of terms from the languages in $\mathbf{D} \downarrow \|\mathbf{T}^*_{\mathcal{L}}\|$ paired with their translations.

The following property states that if τ is in set of translations in $\mathbf{T}^*_{\mathcal{L}}$, then every term t in $dom(\tau)$ actually is mapped to a term in $\mathcal{L}$ by τ.

$$\forall \mathbf{T}\!:\!\mathbf{Tr}\, Set.\ \forall \mathcal{L} \subseteq Term_I.\ \forall \tau \in \mathbf{T}^*_{\mathcal{L}}.\ \forall t \in dom(\tau).\ \tau(t) \in \mathcal{L}$$

The proof of this property is by induction on the level k at which τ occurs in $\mathbf{T}^*_{\mathcal{L}}$ and then follows directly from the definition and the properties of composition.

The fact that translations are not necessarily invertible determines how search is done in the languages that are one or more translation steps from $\mathcal{L}$; we apply the search methods implemented for $\mathcal{L}$ to terms in $\langle t, \tau \rangle \in \mathbf{T}^*_{\mathcal{L}}$ by searching against the translated term $\tau(t)$.

As an example of these definitions, consider the following. There are extant translations of HOL terms to classical Nuprl terms (τ_1), a translation of Isabelle terms to classical Nuprl terms (τ_2) and a translation of ACL2 terms into HOL terms (τ_3).

$$\|\{\}^0_{Nuprl})\| = \{Nuprl\}$$
$$\|\{\tau_1, \tau_2, \tau_3\}^1_{Nuprl}\| = \{HOL, Isabelle\}$$
$$\|\{\tau_1, \tau_3\}^1_{Nuprl}\| = \{HOL\}$$
$$\|\{\tau_1, \tau_2, \tau_3\}^2_{Nuprl}\| = \{ACL2\}$$
$$\|\{\tau_1, \tau_2\}^2_{Nuprl}\| = \{\}$$

Note that the levels as specified here are not cumulative; *e.g.* $Nuprl \in \|\{\tau_1, \tau_3\}^0_{Nuprl}\|$ but $Nuprl \notin \|\{\tau_1, \tau_3\}^1_{Nuprl}\|$. Thus a user interested in searching HOL theorems but excluding theorems of Nuprl to satisfy a Nuprl proof can specify the domain of search as $\|\{\tau_1, \tau_3\}^1_{Nuprl}\|$.

Note that the translations between these different logical theories preserve validity of theorems but do not necessarily translate proofs. Translations are justified somehow, formally or informally. But such justifications may be based on semantic arguments and the translation of proofs is unknown.

3.5 Deductive Searching

Based on these ideas we propose the following general framework for search in heterogeneous databases of theorems from multiple logics. We cast our description in terms of sequents, though it should be obvious how to recast these ideas in non-sequent based logics.

We are interested in searching the library to complete a proof of some sequent of the form $\Gamma \vdash_{\mathcal{L}} \Delta$. Should some $\phi \in \Delta$ already be proved in $\mathcal{L}$ and stored in the library, then $\Gamma, \phi \vdash_{\mathcal{L}} \Delta$ can be trivially proved in $\mathcal{L}$ by cutting in ϕ and then invoking the axiom rule. Less directly, perhaps there is some translation mapping a theorem of some other logic into the term ϕ in the language of $\mathcal{L}$.

Search will be performed using procedures that are, in most cases, incomplete. Our framework assumes that a search procedure used to find results within the

context of some logic $\mathcal{L}$ can construct a "proof" in $\mathcal{L}$ when a search is successful *e.g.* a search procedure to be used in the context of an HOL theorem will return both the lemmas found in the database *and* tactic to apply them in the context of sequent being searched for.

Let $\Gamma \vdash_{\mathcal{L}} \Delta$ be a sequent in the logical language $\mathcal{L}$, let $\mathbf{D}$ be a collection of databases $\{\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n\}$ and let $\mathcal{S}$ be a proof search procedure for $\mathcal{L}$. We define $[\![\Gamma \vdash_{\mathcal{L}} \Delta]\!]_{\mathbf{D},\mathcal{S}}$ as follows:

$$[\![\Gamma \vdash_{\mathcal{L}} \Delta]\!]_{\mathbf{D},\mathcal{S}} \stackrel{def}{=} \{\langle \Gamma', \rho \rangle | \Gamma' \subseteq \cup\mathbf{D} \wedge \rho \text{ proves } \Gamma, \Gamma' \vdash_{\mathcal{L}} \Delta\}$$

where $\cup\mathbf{D}$ is the set of all terms in the databases in the set $\mathbf{D}$. Thus $[\![\Gamma \vdash \Delta]\!]_{\mathbf{D},\mathcal{S}}$ is a set of pairs consisting of lists of theorems Γ' from the databases in $\mathbf{D}$, paired with a method of proof ρ which proves the sequent $\Gamma, \Gamma' \vdash_{\mathcal{L}} \Delta$. The proof ρ (together with the theorems in Γ') is the information needed by the prover for the logic $\mathcal{L}$ to complete the proof of the sequent $\Gamma \vdash_{\mathcal{L}} \Delta$. We write $[\![\Delta]\!]_{\mathbf{D},\mathcal{S}}$ for $[\![\vdash \Delta]\!]_{\mathbf{D},\mathcal{S}}$ and $[\![\phi]\!]_{\mathbf{D},\mathcal{S}}$ for $[\![\vdash \phi]\!]_{\mathbf{D},\mathcal{S}}$.

Now we discuss some consequences and applications of the definition.

Typically, the actual answer set $[\![\Gamma \vdash_{\mathcal{L}} \Delta]\!]_{\mathbf{D},\mathcal{S}}$ is infinite; to see this note that once some list of terms is enough to prove the desired result, any extension of that list will also do[13]. However, note that non-empty approximations to $[\![\Gamma \vdash_{\mathcal{L}} \Delta]\!]_{\mathbf{D},\mathcal{S}}$ are usually satisfactory answers to queries *i.e.* any answer provides a means to prove $\Gamma \vdash_{\mathcal{L}} \Delta$ from the contents of databases in $\mathbf{D}$. Indeed, although $[\![\Gamma \vdash_{\mathcal{L}} \Delta]\!]_{\mathbf{D},\mathcal{S}}$ is defined as a complete answer, only one answer is ever required to discharge the proof obligation. Multiple answers may provide the requester with options allowing them to make choices based on any number of criteria. We can imagine that one criteria might be to choose the answer that requires the minimum update to the local database. Others might be based on elegance.

To search a collection of databases $\mathbf{D}$ for an individual theorem ϕ, one searches for an approximation of $[\![\phi]\!]_{\mathbf{D},\mathcal{S}}$. Note that if any theorem ϕ of $\mathcal{L}$ is in the database, then $\langle \{\phi\}, Axiom\{\phi\} \rangle \in [\![\phi]\!]_{\mathbf{D},\mathcal{S}}$ where $Axiom\{\phi\}$ is the axiom rule for $\mathcal{L}$ *i.e.* the rule justifying sequents of the form

$$\Gamma_1, \phi, \Gamma_2 \vdash_{\mathcal{L}} \Delta_1, \phi, \Delta_2$$

Thus, we have defined a framework for deductive search in a way that users can both account for the results they receive and can apply the results in proofs.

Name and Content based Search in the Deductive Framework We note here that name and content based searches can be fit into the framework just described for deductive search. For name searches, we assume that there is a function *name* mapping library objects to user specified names (strings of characters) and returning the empty string if a name does not exist. We define a logic of names $\mathcal{L}N$, where the language of the logic of names is $Term_I$ (all

[13] Of course we are excluding various resource-bounded and substrutural logics from this consideration.

terms, including strings, are in the language of the logic of names). The logic $\mathcal{L}N$ has one proof rule.

$$\frac{}{t \vdash_{\mathcal{L}N} s}\,\text{Ax} \qquad \text{if } s \in \mathit{string} \wedge\ s \subseteq \mathit{name}(t)$$

Here, s is a string and $s \subseteq s'$ if and only if s is a substring of s'. Thus, a name search for all objects in some collection of databases $\mathbf{D}$ is computed as $[\![s]\!]_{\mathbf{D},\mathcal{L}N}$. To search the names of the terms of some language $\mathcal{L}$ for a particular string s can be specified as $[\![s]\!]_{D\downarrow\mathcal{L},\mathcal{L}N}$; *e.g.* to search Nuprl terms having the string "*list*" as a substring of their name is specified as $[\![\text{“}\mathit{list}\text{”}]\!]_{(\mathbf{D}\downarrow \mathit{Nuprl}),\mathcal{L}N}$. The result of the search would be a set of pairs $\{(t_1, Ax), \cdots, (t_n, Ax)\}$.

We can cast content-based search in the deductive framework by similarly defining a logic of content.

4 Apologia and Conclusion

In this paper we have described an implementation of a peer-to-peer framework for connecting databases of formalized mathematics [17] and the term structures used to communicating between them. We have proposed a framework for deductive searching in distributed collections of heterogeneous databases and have described how name and content based searchers can be cast into the deductive framework. We have emphasized that both *evidence* and *effective methods* of translation and proof should be included as part of the results of searches. There is obviously significant work that remains to be done, most features of the proposed framework have not been implemented. Work on representing evidence using Allen's certificate mechanism continues at Cornell and Wyoming. We have only implemented name and content based searching and intend to further explore more powerful deductive methods based on heuristic search.

In a number of ways this paper is unsatisfactory: some aspects of the proposed framework for sharing and searching have been elaborated in too much detail; while a number of aspects of the presentation are too vague. However, we believe that the proposed approach has several advantages. We do not propose to impose any particular logic or any absolute criteria for correctness on users. To us, any attempt to make such impositions will result in failure, perhaps not for technical reasons but for social ones[14]. Choice of logic and the criteria for correctness are matters for individual deliberation. Instead, we have proposed a framework within which mechanisms for translating between logics can be implemented and where mechanisms to account for results is embedded within the framework. The only imposition we reluctantly make is one of syntax, of term structure. And although we can imagine that XML or some other structured notation would work as well as the one presented here, we can not imagine how to avoid such an imposition. In any case, matters of syntax require far less commitment than

[14] One might reasonably claim that the QED project [4] ended prematurely down for precisely this reason.

matters of semantics. We believe that something very much like the system proposed here, if not this one, will eventually provide a practical means for seamless sharing formal mathematics.

Acknowledgments We would like to thank Stuart Allen and Constable at Cornell for the intellectual enjoyment gained from the time spent discussing FDL related matters. We thank Rich Eaton also at Cornell for cheerful responses to our unreasonable requests for his time and for his programming support. The first author also thanks John Paul at the University of Wyoming for acting as a sounding board for many of the ideas presented here.

References

1. Stuart Allen. *Nuprl Basics.* Cornell University, 2001. `http://www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/`.
2. Stuart Allen. Abstract identifiers, intertextual reference and a computational basis for recordkeeping. *First Monday*, 9(2), February 2004. `http://firstmonday.org/issues/issue9_2/allen/`.
3. Stuart F. Allen, Mark Bickford, Robert Constable, Richard Eaton, and Christoph. Kreitz. A Nuprl-PVS connection: Integrating libraries of formal mathematics. Technical Report TR2003-1889, Cornell University, 2003. `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1889`.
4. Anonymous. QED Manifesto. `http://www-unix.mcs.anl.gov/qed/`.
5. James Caldwell and Judith Underwood. Classical tools for constructive proof search. In Didier Galmiche, editor, *Proceedings of the CADE-13 Workshop on Proof search in Type-theoretic languages.*, Rutgers N.J., July 1996.
6. James Caldwell and Tjark Weber. A formal framework for constructive category theory. `http://www.cs.uwyo.edu/~jlc/papers`, July 2003.
7. Gilles Dowek. *Higher-Order Unification and Matching*, chapter 16, pages 1009–1065. In Robinson and Voronov [23], 2001.
8. The Formal Digital Libraries Project (Homepage). `http://www.nuprl.org/html/Digital_Libraries.html`.
9. D. M. Gabbay and M. de Rijke, editors. *Frontiers of Combining Systems 2 (Proceedings of the Second International Workshop, FroCoS'98, Amsterdam, The Netherlands, October 1998)*, volume 7 of *Studies in Logic and Computation.* Research Studies Press Ltd., 2000.
10. Joseph Goguen and Rod Burstall. Introducing institutions. In Edward Clarke and Dexter Kozen, editors, *Proceedings, Logics of Programming Workshop*, volume 164 of *LNCS*, pages 221–256. Springer, 1984.
11. Joseph A. Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
12. Jim Grundy. Trustworthy storage and exchange of theorems. Technical Report TUCS-TR-1, Turku, Finland, April 1996.
13. Jaakko Hintikka. *The Principles of Mathematics Revisited.* Cambridge University Press, 1996.

14. Douglas Howe. Toward sharing libraries of mathematics between theorem provers. In Gabbay and de Rijke [9], pages 161–176.
15. Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the* 11th *International Conference on Theorem Proving in Higher Order Logics*, volume 1125, of *LNCS*, pages 267–282. Springer-Verlag, 1996.
16. G. Huet and A. Saibi. Constructive category theory. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT, 1998.
17. Christoph Jechlitschek. *Distributed Sharing of Formalized Mathematics: a P2P approach*. Master's thesis, University of Wyoming, Laramie, WY, May 2004.
18. The JXTA project homepage. `http://www.jxta.org`.
19. Evan Moran. Forthcoming Cornell Ph.D. Thesis, Dept. of Computer Science.
20. Pavel Naumov. Importing Isabelle formal mathematics into Nuprl. *The 12th International Conference on Theorem Proving in Higher Order Logics, supplemental proceedings*, 1999. `http://www-sop.inria.fr/croap/TPHOLs99/ps/paper4.ps`.
21. Pavel Naumov, Mark O. Stehr, and Jose Meseguer. The HOL/NuPRL proof translator: A practical approach to interoperability. In *Proceedings of the* 14th *International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *LNCS*, pages 329 – 345. Springer, 2001.
22. National Center for Biotechnology Information (Homepage). `http://www.ncbi.nlm.nih.gov/`.
23. Alan Robinson and Andrei Voronov, editors. *Handbook of Automated Reasoning: Volume II*. MIT, North Holland, 2001.
24. R. Sekar, I. V. Ramakishnan, and Ardrei Voronkov. *Term Indexing*, chapter 26, pages 1855–1964. In Robinson and Voronov [23], 2001.
25. Mark Staples. Linking ACL2 and HOL. Technical Report 476, Cambridge University, Computer Laboratory, 1999. `http://citeseer.ist.psu.edu/staples99linking.html`.
26. Andrzej Tarlecki. Towards heterogeneous specifications. In Gabbay and de Rijke [9], pages 337–360.

Mechanical Verification of Automatic Synthesis of Failsafe Fault-Tolerance[1]

(Extended Abstract)

Sandeep S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir

Department of Computer Science and Engineering,
Michigan State University,
48824 East Lansing, Michigan, USA
{sandeep, borzoo, ebnenasi}@cse.msu.edu
http://www.cse.msu.edu/~{sandeep,borzoo,ebnenasi}

Abstract. Fault-tolerance is a crucial property in many systems. Thus, mechanical verification of algorithms associated with synthesis of fault-tolerant programs is desirable to ensure their correctness. In this paper, we present the mechanized verification of the algorithm that automates the synthesis algorithm for adding failsafe fault-tolerance to a given fault-intolerant program using the PVS theorem prover. By this verification, not only we prove the correctness of the synthesis algorithm, but also we guarantee that any program synthesized by this algorithm is correct by construction. Towards this end, we formally define a framework for formal specification and verification of fault-tolerance that consists of abstract definitions for programs, specifications, faults, and levels of fault-tolerance, so that they are independent of platform and architecture. The essence of the synthesis algorithm involves fixpoint calculations. Hence, we also develop a reusable library for fixpoint calculations on finite sets in PVS.

Keywords: Fault-tolerance, PVS, Program synthesis, Program transformation, Mechanical verification, Theorem proving, Addition of fault-tolerance

1 Introduction

Fault-tolerance is a necessity in most computer systems and, hence, one needs strong assurance of fault-tolerance properties of a given system. Mechanical verification of such systems is one way to get a strong form of assurance. The related work in the literature has focused on verification of concrete fault-tolerant programs. For example, Owre et al [1] present a survey on formal verification of

[1] Extended version appears in the proceedings of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04) Verona, Italy.
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

a fault-tolerant digital-flight control system. Mantel and Gärtner [2] verify the correctness of a fault-tolerant broadcast protocol. Qadeer and Shankar [3] mechanically verify the self-stability property of Dijkstra's mutual exclusion token ring algorithm [4]. Kulkarni, Rushby, and Shankar [5] verify the same algorithm by exploiting the theory of detectors and correctors [6].

While the verifications performed in [1–3, 5] enable us to gain confidence in the programs being verified, it is difficult to extend these verifications to other programs. A more general approach, therefore, is to verify algorithms that generate fault-tolerant programs.

With this motivation, in this paper, we focus on the problem of *verifying an algorithm that synthesizes fault-tolerant programs*. With such verification, we are guaranteed that all the programs generated by the synthesis algorithm indeed satisfy their fault-tolerance requirements. Towards this end, we verify the transformation algorithm for adding failsafe fault-tolerance, presented by Kulkarni and Arora [7] using the PVS theorem prover. To verify this algorithm, first, we model a framework for fault-tolerance in PVS. This framework consists of definitions for programs, specifications, faults, and levels of fault-tolerance. Then, we verify that any program synthesized by the algorithm is indeed failsafe fault-tolerant. By this verification, we ensure that any program synthesized by this algorithm is also correct by construction and, hence, there is no need to verify the individual synthesized programs.

We note that the algorithms in [7], are the basis for their extensions to deal with simultaneous occurrence of multiple faults from different types [8] and for synthesizing distributed programs [9,10]. Thus, the specification and verification of transformation algorithms in [7] is reusable in developing specification and verification of algorithms in [8–10]. Since fixpoint calculation is at the heart of the synthesis algorithm for adding failsafe, we also develop a library for fixpoint calculations on *finite sets* in PVS. This library is reusable for other purposes that involve fixpoint calculations as well.

Contributions of the paper. The contributions of this paper are as follows: (1) We verify the correctness of the synthesis algorithm for adding failsafe fault-tolerance in [7]. Thus, not only we ensure its correctness but also we guarantee that any program synthesized by the algorithm is also correct by construction. (2) We provide a foundation for formal specification and verification of later research work that are extensions of [7]. (3) We develop a reusable library in PVS for fixpoint calculations on finite sets.

Organization of the paper. The organization of the paper is as follows: We provide the formal definitions of programs, specifications, faults, and fault-tolerance in Section 2. Using these definitions, we formally state the problem of mechanical verification of synthesis of failsafe fault-tolerant programs in Section 3. In Section 4, first, we develop a theory for fixpoint calculations on finite sets. Then, based on the definitions in Section 2 and our fixpoint calculation library, we formally specify the synthesis algorithm for adding failsafe tolerance in PVS. In Section 5, we present verification of the algorithm for synthesizing failsafe fault-tolerant programs. Finally, we make concluding remarks and discuss future work in Section 6.

2 Modeling a Fault-Tolerance Framework

In this section, we give formal definitions for programs, specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [11]. The definitions of faults and fault-tolerance are adapted from Arora and Gouda [12] and Kulkarni [6]. We also discuss how we model the definitions in PVS in an abstract way, so that they are independent of any particular program.

2.1 Program

A program p is a finite set of transitions in its state space. In our framework, the notion of state is abstract. Hence, in PVS, we model state by an UNINTERPRETED TYPE [13]. Likewise, a transition is modeled as an ordered pair of states, which is also an uninterpreted type. We also assume that the number of states and transitions are finite. The state space of p, S_p, is the set of all possible states of p. In PVS, we model the state space by the finite *fullset* over states.

We model program, p, by a subset of $S_p \times S_p$. A state predicate of p is a subset of S_p. In PVS, we model a state predicate, StatePred, as a finite set over states. The type Action denotes finite sets of transitions. A state predicate S is closed in the program p iff for all transitions (s_0, s_1) in p, if $s_0 \in S$ then $s_1 \in S$. Hence, we define closure as follows: $closed(S,p) = (\forall s_0, s_1 \mid (s_0, s_1) \in p : (s_0 \in S \Rightarrow s_1 \in S))$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff any pair of two consecutive states is a transition in p. We formalize this by a DEPENDENT TYPE [13] as follows:

$$Computation(p) : TYPE = \{c : sequence[state] \mid (\forall i \mid i \geq 0 : (c_i, c_{i+1}) \in p)\}$$

where $sequence[state] : \mathbb{N} \rightarrow state$ and p is any finite set of type Action. A computation prefix is a finite sequence of states, where the first j steps are transitions in the given program:

$$prefix(p, j) : TYPE = \{c : sequence[state] \mid (\forall i \mid i < j : (c_i, c_{i+1}) \in p)\}$$

We deliberately model computation prefixes by infinite sequences of which only a finite part is used. This is due to the fact that using finite sequences in PVS is not very convenient and the type checker generates several proof obligations whenever finite sequences are used.

The projection of program p on state predicate S consists of transitions of p that start in S and end in S, denoted as $p \mid S$. Similar to the notion of program, we model projection of p on S by a finite set of transitions: $p \mid S = \{(s_0, s_1) \mid (s_0, s_1) \in p \ \wedge \ (s_0, s_1 \in S)\}$.

2.2 Specification

The specification consists of a safety specification and a liveness specification. The safety specification is specified as a set of *bad transitions*. Thus, for program p,

its safety specification is a subset of $S_p \times S_p$. Hence, we can model the safety specification by a finite set of transitions, called *spec*. We explain the liveness issue in Section 2.3.

Given program p, state predicate S, and specification *spec*, we say that p satisfies its specification from S iff (1) S is closed in p, and (2) every computation of p that starts in a state where S is true, does not contain a transition in *spec*. If p does not satisfy its specification from S, we say p violates its specification. If p satisfies specification from S and $S \neq \{\}$, we say that S is an invariant of p. Since we do not deal with a specific program, in PVS, we model an invariant by an arbitrary state predicate that is closed in p.

2.3 Faults and Fault-Tolerance

The faults that a program is subject to are systematically represented by a finite set of transitions. A class of fault f for program p is a subset of $S_p \times S_p$. A computation of program p in presence of faults f is an infinite sequence of states where either a transition of p or a transition of f occurs at every step. Hence, we model computation of program in presence of faults as $c : Computation(p \cup f)$.

We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$, and (2) T is closed in $p \cup f$. Thus, we model fault-span in PVS as follows: $FaultSpan(T, S, p \cup f) = ((S \subseteq T) \wedge (closed(T, p \cup f)))$. Observe that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the transitions in f. Hence, we define the different levels of fault-tolerance based on the behavior of the fault-tolerant program in its fault-span.

We say that p is failsafe f-tolerant (read as fault-tolerant) to its specification from S iff two conditions hold: (1) p satisfies its specification from S, and (2) there exists T such that T is an f-span of p from S, and no prefix of a computation of $p \cup f$ that starts in T has a transition in *spec*.

In [7], the liveness specification is modeled implicitly. Specifically, for failsafe fault-tolerance, the requirement is that the fault-tolerant program does not *deadlock* in the absence of faults. A program deadlocks in state s_0 iff $\forall s_1 \mid s_1 \in S : (s_0, s_1) \notin p$.

3 Problem Statement

In this section, we formally state the the problem of automatic synthesis of failsafe fault-tolerance. As described in Section 2, the fault-intolerant program p is specified in terms of its state space S_p, its transitions, p, and its invariant, S. The specification provides a set of bad transitions (that should not occur in program computation). The faults, f, are specified in terms of a finite set of transitions. Likewise, the fault-tolerant program p' is specified in terms of its state space S_p, its set of transitions, say p', its invariant, S', its specification, *spec*, and the type of fault-tolerance it provides.

The Transformation Problem
Given p, S, *spec*, and f such that p satisfies *spec* from S.
Identify p' and S' such that:
$S' \subseteq S$
$(p'|S') \subseteq (p|S')$
p' is failsafe f-tolerant to *spec* from S'

We now explain the reasons behind the first two conditions briefly:

- If S' contains states that are not in S then, in the absence of faults, p' will include computations that start outside S and hence, p' contains new behaviors in the absence of faults. Therefore, we require that $S' \subseteq S$.
- Regarding the transitions of p and p', we focus only on the transitions of $p'|S'$ and $p|S'$. If $p'|S'$ contains a transition that is not in $p|S'$, p' can use this transition in a new computation in the absence of faults and hence, we require that $p'|S' \subseteq p|S'$.

Soundness. An algorithm for the transformation problem is sound iff for any given input, its output, namely p' and S', satisfies the transformation problem.

Our goal is to mechanically verify that the proposed algorithm in [7], for adding failsafe fault-tolerance is indeed sound.

4 Description and Specification the Synthesis Algorithm

In this section, we describe the synthesis algorithm for adding failsafe fault-tolerance proposed in [7], and explain how we formally specify it in PVS. The essence of adding failsafe fault-tolerance to a given fault-intolerant program is re-calculation of the invariant of the fault-intolerant program which in turn involves calculating the fixpoint of a formula. More specifically, we calculate fixpoint of a given formula to (i) calculate the set of states from where safety may be violated by faults alone; (ii) remove deadlock states that occur in a given set of states.

The $\mu-$calculus theory of the PVS prelude contains general definitions of the standard fixpoint calculation, however, it is not convenient to use that theory in the context of our problem. This is due to the fact that this library focuses on infinite sets and is not specialized to account for the properties of functions used in the synthesis of fault-tolerant programs. By contrast, we find that by customizing the theory to the properties of functions used in the synthesis of fault-tolerant programs, we can simplify the verification of the synthesis algorithm. Hence, in Section 4.1, we develop a theory for fixpoint calculations on *finite sets* and we verify it in Section 5.1. This theory is expected to be reusable for other formalizations that involve fixpoint calculations on finite sets. Based on the definitions in Section 4.1, we model the synthesis algorithm for addition of failsafe fault-tolerance in Sections 4.2.

4.1 Specification of Fixpoint Calculation for Finite Sets

In this section, we describe how we formally specify fixpoint calculation for finite sets in PVS. A fixpoint of a function $f : X \to X$ is any value $x_0 \in X$ such that $f(x_0) = x_0$. In the context of finite sets, domain and range of f, X, are both finite sets of finite sets. Throughout this section and in Section 5.1, the variables i, j, k range over natural numbers. The variable x is any finite set of any uninterpreted type. Variable b is any member of such finite set.

One type of functions used in synthesis of failsafe fault-tolerance is a decreasing function for which the largest fixpoint is calculated. Towards this end, we start from an *initial set* and at each step of calculation, we remove a subset of the initial set that has a certain property. Thus, the type DecFunc is the type of functions g, such that $g : \{A : finiteset\} \to \{B : finiteset \mid B \subseteq A\}$. In other words, for all finite sets x, $g(x) \subseteq x$. With such a decreasing function, we define $Dec(i, x)(g)$ to formalize the recursive behavior of the largest fixpoint calculation. $Dec(i, x)(g)$ keeps removing the elements of the initial set, x, that the function g of type DecFunc returns at every step:

$$Dec(i, x)(g) = \begin{cases} Dec(i-1, x)(g) - g(Dec(i-1, x)(g)) & \text{if } i \neq 0; \\ x & \text{if } i = 0 \end{cases}$$

We define the largest fixpoint as follows:

$$LgFix(x)(g) = \{b \mid \forall k : b \in Dec(k, x)(g))\}$$

Our goal is to prove the following property of largest fixpoint based on our definitions:

$$g(LgFix(x)(g)) = \emptyset$$

Likewise we define an increasing function, r, for which the smallest fixpoint is calculated:

$$Inc(i, x)(r) = \begin{cases} Inc(i-1, x)(r) \cup r(Inc(i-1, x)(r)) & \text{if } i \neq 0; \\ x & \text{if } i = 0 \end{cases}$$

Finally, we define the smallest fixpoint as follows:

$$SmFix(x)(r) = \{b \mid \exists k : b \in Inc(k, x)(r)\}$$

4.2 Specification of the Synthesis of Failsafe Fault-Tolerance

The essence of adding failsafe tolerance is to remove the states from where safety may be violated by one or more fault transitions. We reiterate the algorithm *Add_failsafe* (from [7]) in Figure 1.

Throughout this section and Sections 5.2, the variables x, s, s_0, s_1 range over states. The variables i, j, k, m range over natural numbers. The variable X ranges over StatePred and the variable Z ranges over Action. As defined in Section 3, p and p' are respectively fault-intolerant and fault-tolerant programs, S and S' are respectively invariants of fault-intolerant and fault-tolerant programs, f is the finite set of faults, and *spec* is the finite set of bad transitions that represents the safety specification.

```
Add_failsafe(p, f : transitions, S : state predicate, spec : specification)
{
        ms := smallestfixpoint(X = X ∪ {s0 | (∃s1 :
                                (s0, s1) ∈ f)  ∧  (s1 ∈ X ∨ (s0, s1) violates spec) };
        mt := {(s0, s1) : ((s1 ∈ ms) ∨ (s0, s1) violates spec) };
        S' := ConstructInvariant(S − ms, p − mt);
        if (S' = {}) declare no failsafe f-tolerant program p' exists;
        else p' :=ConstructTransitions(p − mt, S')
}

ConstructInvariant(S : state predicate, p : transitions)
// Returns the largest subset of S such that computations of p
                                within that subset are infinite
    return largestfixpoint(X = (X ∩ S) − {s0 | (∀s1 : s1 ∈ X : (s0, s1) ∉ p)}

ConstructTransitions(p : transitions, S : set of states)
    { return p − {(s0, s1) : s0 ∈ S  ∧  s1 ∉ S} }
```

Fig. 1. The synthesis algorithm for adding failsafe tolerance

In order to construct ms, the set of states from where safety can be violated by one or more fault transitions, first, we define $msInit$ as the finite set of states from where safety can be violated by a *single* fault transition. Note that $(s_0, s_1) \in spec$ means violation of the safety specification. Formally,

$$msInit : StatePred = \{s_0 \mid \exists\ s_1\ :\ (s_0, s_1) \in f\ \wedge\ (s_0, s_1) \in spec\}$$

Now, we define a function, *RevReachStates*, that calculates a state predicate from where states of another finite set, rs, are reachable by fault transition. Formally,

$$RevReachStates(rs : StatePred) : StatePred = \{s_0 \mid \exists\ s_1 : s_1 \in rs \wedge (s_0, s_1) \in f \wedge s_0 \notin rs\}$$

We use the definition of smallest fixpoint in Section 4.1 to define the state predicate ms. Towards this end, we instantiate the initial set with *msInit*, and the r function with *RevReachStates*:

$$ms : StatePred = SmFix(msInit)(RevReachStates)$$

Then, we define the finite set of transitions, mt, that must be removed from p. These transitions are either transitions that may lead a computation to reach a state in ms or transitions that directly violate safety:

$$mt : Action = \{(s_0, s_1) \mid (s_1 \in ms \vee (s_0, s_1) \in spec)\}$$

The algorithm *Add_failsafe* removes the set ms from the invariant of the fault-intolerant program S. However, this removal may create deadlock states. The set of deadlock states in ds of program Z is denoted as follows:

$$DeadlockStates(Z)(ds : StatePred) : StatePred = \{s_0 \mid s_0 \in ds : (\forall s_1 \mid s_1 \in ds : (s_0, s_1) \notin Z)\}$$

We construct the invariant of the fault-tolerant program by removing the deadlock states to ensure that computations of fault-tolerant program are infinite

(cf. Section 2.3). In general, we define $ConstructInvariant$ using largest fixpoint of a finite set X, that removes deadlock states of a given state predicate X:

$$ConstructInvariant(X, Z) : StatePred = LgFix(X)(DeadlockStates(Z))$$

The formal definition of the invariant of fault-tolerant program is as follows:

$$S' : StatePred = ConstructInvariant(S - ms, p - mt)$$

Finally, we construct the finite set of transitions of fault-tolerant program by removing the transitions that violate the closure of S':

$$p' : Action = p - mt - \{(s_0, s_1) \mid ((s_0, s_1) \in (p - mt)) \wedge (s_0 \in S' \wedge s_1 \notin S')\}$$

5 Verification of the Synthesis Algorithm

In this section, we verify the soundness of the synthesis algorithm for adding failsafe fault-tolerance based on the formal specification in Section 4.

5.1 Verification of the Fixpoint Theory

In order to verify the soundness of the synthesis algorithm for adding failsafe fault-tolerance, first, we prove the properties of fixpoint calculations (cf. Section 4.1) in Theorem 5.6.

Lemma 5.1: Until the fixpoint is achieved, the cardinality of $Dec(j+1, x)$ is less than or equal to $|x| - j - 1$. Formally,

$$\forall j : ((g(Dec(j, x)(g)) \neq \emptyset) \implies |(Dec(j+1, x)(g)| \leq |x| - j - 1))$$

Proof. We prove this lemma by induction on j. In the base case, $j = 0$, after eliminating the quantifiers and expanding the definitions, we need to show if $g(x)$ is nonempty then $|x - g(x)| \leq |x| - 1$. We prove this by using two predefined lemmas in PVS: $\forall y, z : finiteset : ((y \subseteq z) \implies (|z - y| = |z| - |y|))$, and $\forall y : finiteset : (y \neq \emptyset \iff |y| > 0)$. After instantiations, using the facts $g(x) \subseteq x$ and $g(x) \neq \emptyset$, the GRIND strategy [14] discharges the base case. For induction step, after eliminating quantifiers, and expanding definitions, we need to prove $(g(Dec(j+1, x)(g)) \neq \emptyset \wedge |Dec(j+1, x)(g)| \leq |x| - j - 1) \implies (|Dec(j+1+1, x)(g)| \leq |x| - (j+1) - 1)$. We discharge the induction step this in the same way we proved the base case. □

Lemma 5.2: If the fixpoint is reached by step j then in any subsequent steps, fixpoint will be maintained. Formally,

$$\forall j : ((g(Dec(j, x)(g)) = \emptyset) \implies (\forall k \mid k \geq j : g(Dec(k, x)(g)) = \emptyset))$$

Proof. After skolemization to remove the universal quantifier, we place induction on k. The base case, $k = j = 0$, is trivially true. In the induction step, we need to prove $(g(Dec(k, x)(g)) = \emptyset) \implies (g(Dec(k+1, x)(g)) = \emptyset)$. By expanding the definition of Dec in the deducing part, $Dec(k+1, x)(g) = Dec(k, x)(g) - g(Dec(k, x)(g))$, and considering the assuming part we infer that $g(Dec(k, x)(g)) = \emptyset$, therefore $g(Dec(k+1, x)(g)) = g(Dec(k, x)(g))$, which is

equal to the empty set. □

Lemma 5.3: There exists a step i such that subsequent applications of g returns the empty set. Formally, $\exists i : (\forall n \mid n \geq i : g(Dec(n, x)(g)) = \emptyset)$
Proof. First, we instantiate i with $|x|$. Then, after skolemization, we need to prove $g(Dec(n, x)(g)) = \emptyset$. Using Lemma 5.1 and instantiating j with $|x|$, we need to show two subgoals:
Subgoal 1: $|Dec(|x| + 1, x)(g)| > |x| - |x| - 1$, which is trivially true.
Subgoal 2: $(g(Dec(|x|, x)(g)) = \emptyset) \implies (g(Dec(n, x)(g)) = \emptyset)$. From Lemma 5.2, we know $\forall j : (g(Dec(j, x)(g)) = \emptyset) \implies (\forall k \mid k \geq j : g(Dec(k, x)(g)) = \emptyset)$. After automatic instantiations, we need to prove $(\forall k \mid k \geq |x| : g(Dec(k, x)(g)) = \emptyset) \implies (g(Dec(n, x)(g)) = \emptyset)$. Manual instantiation of k with n discharges the lemma. □

Lemma 5.4: There exists a step j where fixpoint is achieved. Formally,
$\exists j : (\forall k \mid k \geq j : ((Dec(k, x)(g) = Dec(j, x)(g)) \ \wedge \ (g(Dec(k, x)(g)) = \emptyset)))$
Proof. Proof of the second conjunct is exactly the same as proof of Lemma 5.3, so we proceed with the proof of the first conjunct. From Lemma 5.3, we know that the existence of j such that $\forall k \mid k \geq j : g(Dec(k, x)(g)) = \emptyset$. Using Lemma 5.3 and after skolemization, we place induction on k. In the base case, $k = j = 0$, we need to show $Dec(0, x)(g) = Dec(j, x)(g)$, which is trivially true. In induction step, we need to prove:

$$\forall i \mid i \geq j : ((Dec(i, x)(g) = Dec(j, x)(g) \wedge g(Dec(i, x)(g)) = \emptyset) \implies (Dec(i + 1, x)(g) = Dec(j, x)(g)))$$

We prove this by applying the rule of extensionality and expanding $Dec(i + 1, x)(g)$, which is equal to $Dec(i, x)(g) - g(Dec(i, x)(g))$. As $g(Dec(i, x)(g)) = \emptyset$, $Dec(i + 1, x)(g) = Dec(i, x)(g) = Dec(j, x)(g)$ and the proof is complete. □

Lemma 5.5: For some value j, $Dec(j, x)$ will reach a fixpoint, and at this step value of $Dec(j, x)$ will be the largest fixpoint. Formally,
$\exists j : (g(Dec(j, x)(g)) = \emptyset \ \wedge \ (Dec(j, x)(g) = LgFix(x)(g)))$
Proof. Similar to proof of Lemma 5.4, the proof of the first conjunct is the same as proof of Lemma 5.3. To prove the second conjunct, first, we apply the rule of extensionality to convert the set equalities to boolean equalities. A propositional split generates two subgoals:
Subgoal 1: $\forall b \in LgFix(x)(g) : b \in Dec(j, x)(g)$. First, we expand the definition of $LgFix = \{b \mid \forall k : b \in Dec(k, x)(g)\}$ in the assuming part. Then, instantiating k with j proves the subgoal.
Subgoal 2: $\forall (b \in Dec(j, x)(g)) : b \in LgFix(x)(g)$.
To verify this subgoal, after expanding the definition of $LgFix$ and eliminating the universal quantifier by skolemization, we need to show $\forall b \in Dec(j, x)(g) : b \in Dec(k, x)(g)$. Using Lemma 5.4, we know that
$\forall i \mid i \geq j : (Dec(i, x)(g) = Dec(j, x)(g) \ \wedge \ g(Dec(i, x)(g)) = \emptyset)$.
We instantiate i with k and by propositional simplification through the GROUND

command [14], we prove this subgoal. □

Theorem 5.6: Application of function g on the largest fixpoint of a finite set returns the empty set. Formally, $g(LgFix(x)(g)) = \emptyset$
Proof. Using Lemma 5.5, the GRIND strategy completes the proof. □

5.2 Verification of the Synthesis of Failsafe Fault-Tolerance

In order to verify the soundness of *Add_failsafe* algorithm, we now prove that the synthesized program, p', satisfies the three conditions of the transformation problem stated in Section 3. More specifically, in Theorems 5.9 and 5.10, we prove the correctness of the first two conditions of the transformation problem. Then, in the remaining theorems, we show that the program synthesized by *Add_failsafe* is indeed failsafe fault-tolerant.

Observation 5.8: $S' \cap ms = \emptyset$
Proof. After expanding the definition of S', $ConstructInvariant$, and $LgFix$, we need to prove: $\forall x : (\forall k : \ x \in Dec(k, S - ms)(DeadlockStates(p - mt)) \implies x \notin ms)$. By instantiating k with 0, propositional simplification discharges the observation. □

Theorem 5.9: $S' \subseteq S$
Proof. Our strategy to prove this theorem is based on the fact that S' is made out of S by removing some states. After expanding the definition of S', $ConstructInvariant$, and $LgFix$, we need to prove:
$\forall k : (\forall x : (x \in Dec(k, S - ms)(DeadlockStates(p - mt)) \implies x \in S))$.
Towards this end, first, we instantiate k with zero. Then, after expanding the definitions, we need to prove $\forall x : (x \in S - ms \implies x \in S)$, which is trivially true. □

Theorem 5.10: $p'|S' \subseteq p|S'$
Theorem 5.11: S' is closed in p'. Formally, $closed(S', p')$
Lemma 5.12: $\forall (s_0, s_1) : ((s_0, s_1) \in f \wedge s_1 \in ms) \implies s_0 \in ms$
Proof. The GRIND strategy discharges this lemma and theorems 5.10 and 5.11. □

Theorem 5.13: $DeadlockStates(p - mt)(S') = \emptyset$
Proof. First, we expand the definitions of S' and $ConstructInvariant$. Then, we need to prove: $DeadlockDtates(p-mt)(LgFix(S-ms)(DeadlockStates(p-mt))) = \emptyset$. Using Theorem 5.6, first, we instantiate x with $LgFix(S - ms)$, and g with $DeadlockStates(p - mt)$ to complete the proof. Then, a sequence of expansions of definitions and propositional simplifications discharge the theorem. □

Remark. Note that Theorem 5.13 is one of the instances where formalization of the fixpoint in Section 4.1 is used. More specifically, $DeadlockStates(p')(S')$ denotes the deadlock states in S' using program p'. We repeatedly remove these

deadlock states. Hence, once the fixpoint is reached, there are no deadlock states.

Lemma 5.14: In the presence of faults, no computation prefix of failsafe tolerant program that starts from a state in S', reaches a state in ms. Formally,

$\forall j : (\forall c : prefix(p' \cup f, j) \mid c_0 \in S' : \forall k \mid k < j : c_k \notin ms)$

Proof. After eliminating the universal quantifier on $c(p' \cup f)$ by skolemization, we proceed by induction on k. In the base case, $k = 0$, we need to prove $c_0 \in S' \Longrightarrow c_0 \notin ms$. The base case can be discharged using Observation 5.8. In induction step, we need to prove $(\forall n \mid n < j : (c_n, c_{n+1}) \in p' \cup f) \Longrightarrow (\forall k \mid k < j : c_k \notin ms \Rightarrow c_{k+1} \notin ms)$. From Lemma 5.12, we know that if the destination of a fault transition , (s_0, s_1), is in ms, then the source, s_0, is in ms as well. This means that if s_0 is not in ms then s_1 is not in ms either. We know that $c_k \notin ms$ and, hence, based on Lemma 5.12, $c_{k+1} \notin ms$. □

Theorem 5.15: Any prefix of any computation of failsafe tolerant program in the presence of faults that starts in S' does not violate safety. Formally,

$\forall j : \forall (c : prefix(p' \cup f), j \mid c_0 \in S') : \forall k | k < j : (c_k, c_{k+1}) \notin spec$

Proof. In Lemma 5.14, we proved that no computation prefix of $p' \cup f$ that starts from a state in S' reaches a state in ms. In addition, p' does not contain any transition that is in *spec*. Thus, a computation prefix of $p' \cup f$ that starts from a state in S' does not contain a transition in *spec*. □

6 Conclusion and Future Work

In this paper, we focused on the problem of verifying transformation algorithm for adding failsafe fault-tolerance that generate fault-tolerant programs that are correct by construction. We would like to note that we have also verified the algorithm for synthesizing (i) *nonmasking* fault-tolerant programs where the program recovers to states from where its specification is satisfied although safety may be violated during recovery, and (ii) *masking* fault-tolerant programs where the program recovers to states from where its specification is satisfied while preserving safety [15, 16].

Since we focus on verification of a transformation algorithm, we note that our results ensure that any program synthesized using the algorithm indeed satisfies its required fault-tolerance properties. Thus, our approach is more general than verifying a particular fault-tolerant program. Also, to verify the algorithm that synthesizes failsafe fault-tolerant programs, we developed a fixpoint library for finite sets. This library is expected to be applicable elsewhere.

In a broader context, the verification of the algorithm considered in this paper will assist us in verifying several other transformations. For example, in [8], the authors extend the algorithms in [7] to deal with multiple classes of faults. The algorithms in [7] have also been used to synthesize fault-tolerant distributed programs. As an illustration, we note that the algorithms in [9, 10, 17] that are extensions of the algorithms in [7] have been used to synthesize solutions for several fault-tolerant programs including, Byzantine agreement, consensus, token

ring, and alternating bit protocol. Thus, the theories developed in this paper are directly applicable to verify the transformation algorithms in [8–10, 17] as well.

References

1. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
2. Heiko Mantel and Felix C.Gärtner. A case study in the mechanical verification of fault-tolerance. Technical Report TUD-BS-1999-08, Department of Computer Science, Darmstadt University of Technology, 1999.
3. S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.
4. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
5. S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA*, pages 33–40, June 1999.
6. S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.
7. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000.
8. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. IEEE Conference on Dependable and Network Systems *(DSN'04)*, 2004.
9. S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.
10. A. Ebnenasir and S. S. Kulkarni. A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~sandeep/software/Code/synthesis-framework/`.
11. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
12. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
13. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001. URL: http://pvs.csl.sri.com/manuals.html.
14. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide: Version 2.4*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001. URL: http://pvs.csl.sri.com/manuals.html.
15. Borzoo Bonakdarpour. Mechanical verification of automatic synthesis of fault-tolerant programs. Master's thesis, Michigan State University, 2004.
16. S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, 2004.
17. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.

ARM6 Formal Verification: Experience with a Commercial Microprocessor

Anthony Fox

University of Cambridge

Abstract. The ARM6 processor has been modelled in HOL at the RTL and ISA levels of abstraction; the entire instruction set has been verified. This paper explains how the models were developed and briefly discusses the verification of the block data transfer and multiply instructions. Exceptions are in the process of being verified – the correctness model with input streams is presented here.

1 Introduction

The ARM6 is a commercial, 32-bit RISC processor that has been widely used in mobile and embedded systems. Section 2 describes how this processor was modelled in HOL, and Section 2.4 discusses the verification of two instruction classes: *block data transfers* and *multiplies*, which are both implemented with *iterative* execute cycles. Section 2.5 presents work in progress – verifying the ARM6 exceptions using a definition of correctness with input streams (Section 3.2). Section 3 contrasts this definition of correctness with the basic version.

1.1 Related Work

Early work on the mechanical verification of processors includes: TAMARACK [18], SECD [12], the partial verification of Viper [6], Hunt's FM8501 [15], and the generic interpreter approach of Windley [30]. Following this work, Miller and Srivas verified some of the instructions of a simple commercial processor called the AAMP5 [22]; this was based on Cyrluk's approach [7].

Recent work has focused on verification techniques applied to complex (but academic) micro-architecture designs, which have out-of-order execution (typically using Tomasulo's algorithm), speculative execution (branch prediction) and exceptions [28, 21, 29, 17, 25, 3, 14]. Most of these projects use variants of the *flushing* correctness model of Burch and Dill [5], which Jones *et al.* extended to out-of-order designs [16]. The instruction set architectures used for academic case studies are usually fairly simple, often based on the DLX architecture of Hennessy and Petterson [13].

Complex commercial designs have also been specified, simulated and verified using ACL2 [4, 19]. There has also been industrial verification work on processor sub-systems; for example, Intel and AMD have verified the IEEE compliance of floating-point hardware [24, 23].

Many notions of correctness have been used in processor verifications and it is not easy to make comparisons; see Aagaard *et. al* [1]. Much work has been built on the flushing approach of Burch and Dill, and bespoke versions have been used in different contexts. However, as Manolios [20] points out, there are some technical problems with this approach.

2 Specification and Verification of the ARM6

The ARM6 specification and verification project, carried out at Leeds and Cambridge, has been funded by the EPSRC. Work initially started at Leeds (Graham Birtwistle, Dominic Pajak and Daniel Schostak) to produce specifications (including ML models) of the ARM architecture (Pajak) and of the ARM6 microprocessor (Schostak). The two students had regular internships with the company and their work was aided with technical data supplied by ARM.[1] In October 2000 work then started at Cambridge, with the aim to verify the processor model. The first work to be carried out at Cambridge was in formalising the correctness framework (Section 3.1) in HOL. This was motivated with a couple of small verification examples: a micro-programmed data path, and a five stage pipeline implementing a minimal instruction set [9, 8].

2.1 The Architecture

Version 4 of the ARM architecture was modelled in HOL [10] – this model has been refined during the course of the project. The specification was influenced by Dominic Pajak's ML model and the standard ARM reference manuals were used as well [11, 27]. Some features of the architecture are listed below:

- It is a 32-bit RISC architecture.
- There are six operating modes and the registers are arranged into overlapping banks. The program counter is register fifteen.
- There is a program status register (CPSR) and five saved versions (SPSR registers).
- All instructions are conditionally executed. The CPSR contains four condition flags.
- There are seven types of exceptions: reset, undefined instruction, software interrupt, prefetch abort, data abort, normal interrupt and fast interrupt.
- There are eight main instruction classes (Table 1) and also coprocessor instructions.

Early on there was some experimentation as to how best to model the underlying data type, 32-bit words, in HOL. A bespoke theory of 32-bit words (using equivalence classes) was eventually developed; the pre-existing HOL theory of words (a list based model developed by Wong [31]) was not really suitable. With the new theory: a word length predicate is not needed; it enables expressions

[1] Dominic and Daniel now work for ARM Ltd. full time.

Table 1. The ARM instruction classes.

Class	Instructions
Branch and Branch with Link	`B`, `BL`
Data Processing	`ADD`, `ADC`, `SUB`, `SBC`, `RSB`, `RSC`, `CMP`, `CMN`, `AND`, `ORR`, `EOR`, `MOV`, `MVN`, `BIC`, `TST`, `TEQ`
Multiply and Multiply Accumulate	`MUL`, `MLA`
PSR Transfer	`MRS`, `MSR`
Single Data Transfer	`LDR`, `STR`
Block Data Transfer	`LDM`, `STM`
Single Data Swap	`SWP`
Software Interrupt and Exceptions	`SWI`

to be readily evaluated (using call-by-value conversion [2]); and it provides an easy mechanism for producing sets of theorems about the logic and arithmetic operations (e.g. addition, multiplication, shifting and bitwise logic). The theory was later generalised for any fixed word length; this is the `n_bit` library, which is included as part of the latest Kananaskis version of HOL.

2.2 The ARM6

The ARM6 is a three stage pipelined processor with a multi-cycled execute stage. A swap instruction, for example, is fetched, decoded and then takes four (or six) cycles to execute. Daniel Schostak's RTL specification [26] was used to produce a HOL model of the ARM6. Schostak produced three specifications of the ARM6, these were: a mathematical style presentation (a set of assignments tagged by instruction step and phase e.g. $t_5\ \phi_2$ is the second phase of the third execute cycle); an engineering style presentation (using a set of tables); and an executable model (ML program). The specifications have three parts: the data path specification, the data path control specification and the pipeline control specification. The mathematical specification is organised by instruction class, instruction step and then by signal order; this enables one to trace the processor's behaviour for a particular instruction type. The engineering specification is organised by pipeline activity and then by signal order; the tables allow one to see how signal behaviour varies according to instruction class and step. Table 1 shows a fragment of the engineering specification for the *pc*-bus write signal, PCWA; this controls whether the program counter register is incremented. Schostak's specifications made distinctions between different types of entities i.e. buses; combinational logic (functional units, multiplexers and static logic); and memory elements (a latch, conditional latch or R-S latch).

The HOL specification of the ARM6 is a hybrid of Schostak's mathematical and engineering specifications. It is organised in accordance with the engineering specification, but with each table converted into an equivalent function (the equivalence is not strict in the case *don't care* output – this simplifies some definitions). Unlike Schostak's specification, no explicit distinction is made between

PCWA

IC	IS	IREG							CPB	
*	*	24	23	21	15	14	13	12	0	
data_proc	t_3	1	0	x	x	x	x	x	x	(true,,n(1111,NBS[6:0]))
data_proc	t_3	x	x	x	1	1	1	1	x	(false,,n(1111,NBS[6:0]))
mrs_msr	t_3	x	x	0	1	1	1	1	x	(false,,n(1111,NBS[6:0]))
...	.	...			...				.	...

Fig. 1. Daniel Schostak's tabular specification of the ARM6.

the different types of entities. The overall cycle level behaviour of the processor is specified using a next state function. Schostak's specification does not define a next state function but the required behaviour can be deduced from the phase and order of presentation of the signal assignments.

The initial HOL processor model left out: hardware interrupts; coprocessor instructions; swaps; multiplies and block data transfers. The design was progressively extended with the inclusion of the swaps, followed by the block transfers and then the multiplies. At each stage the design was verified with respect to an instruction set model which only covered the instructions implemented. This approach enabled working verifications to be completed (and archived) before adding new features which would take some time to verify.

2.3 Abstractions

The correctness of the ARM6 is expressed using data and temporal abstraction maps (Section 3.1). The data abstraction projects out the memory and registers from the processor's state space. The processor's program counter has value $pc + 8$ because it is used for instructions fetch (i.e. it is two instructions, or eight bytes, ahead of the instruction being executed) and the data abstraction accounts for this by subtracting eight. It is shown that the data abstraction is a surjective map from the initial states implementation to the initial (all) states of the specification; this proves that the implementation is not partial (or trivial). The temporal abstraction is defined using a *duration map*: this gives the number of cycles needed to complete instruction execution from a given processor state.

Store instructions require special attention when the memory address is $pc+4$ or $pc + 8$; instruction fetch and decode are invalidated by this localised self modification of code. Two approaches to this were tried before settling on a third solution. The first approach was to block writes to these addresses and the second solution was to 'fix' the processor implementation by ensuring that the pipeline's state is correctly updated. Both of these methods have the disadvantage that they do not reflect the actual ARM6 behaviour. The third method was to modify the ISA model so as to reflect the pipelined behaviour; this was comparatively simple to specify and verify. The data abstraction projects out the opcodes of the fetched and decoded instructions.

2.4 Non-trivial Instruction Classes

Block Data Transfers Block data transfer instructions load/store a set of general purpose register values from/to main memory; the instruction format is shown in Figure 2. These instructions are used for procedure entry and return (saving and restoring workspace registers), and in writing memory block copy routines.

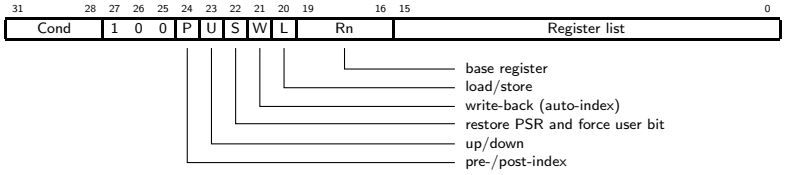


Fig. 2. Block data transfer instruction encoding.

The five option flags (bits 20–24) give us thirty-two possible variants. For example, the instruction `LDMLSDB r10!, {r1,r2,pc}^` is encoded as follows:

$$1001 \cdot 100 \cdot 1 \cdot 0 \cdot 1 \cdot 1 \cdot 1 \cdot 1010 \cdot 1000000000000110$$

Bit `L` is set – this indicates that it is load. Bit `P` is set and bit `U` is clear: this means that the load address (initially base register value `r10`) is decremented before each transfer. Bits `S` and fifteen are set: this means that the SPSR for the current mode is copied to the CPSR. Bit `W` is set: this means that the base address takes the value of the last load address (i.e. subtract twelve). The register set is encoded in the bottom sixteen bits i.e. bits one, two and fifteen are set. Transfers always occur in register index order.

The ARM6 implements this instruction class using a 16-bit mask; this keeps track of which registers have already been transfered and is used to compute the index of the next register to be transfered. The following table shows the value of this mask and the *priority register* (`rp`) for each execute cycle (`is`) of the block load above:

`is`	`mask`	`mask` $\wedge_{16}$ `ireg`	`rp`	`orp`	`oorp`
t_3	1111111111111111	1000000000000110	1		
t_4	1111111111111101	1000000000000100	2	1	
t_n	1111111111111001	1000000000000000	15	2	1
t_n	0111111111111001	0000000000000000	$\perp$	15	2
t_m	$\perp$	$\perp$	$\perp$	$\perp$	15

The t_n instruction step is repeated until the masked value (column three) is zero, there is then a final step t_m. The state of the mask and priority register becomes undefined ($\perp$) but the transfers have been completed by this stage.

The ISA specification creates a list of register indices and then defines the state of the memory (store) or registers (load) by applying an appropriate fold operation over this list. There is, therefore, a significant difference in the way the two models work and some tricky lemmas were needed in order to relate the ISA (list based) and micro-architecture (masking) models.

The state of the processor during each execute cycle is established with the use of invariants. The block data transfers and multiplies were the only classes for which such invariants were required.

Multiplies At the ISA level multiplies are fairly simply but the ARM6 implementation is quite complex. Unlike most modern microprocessors, the ARM6's ALU cannot carry out multiplication directly. Instead, the instruction class is implemented using ALU addition/subtraction and the barrel shifter (which shifts the value on the data path's B bus). The processor's control logic is used to implement the *modified Booth's algorithm*; this can take from two to seventeen execute cycles to complete. The output of the ALU on each cycle is:

$$\mathrm{ALU6}^{*}(\mathit{borrow2}, \mathit{mul}, \mathit{alua}, \mathit{alub}) = \begin{cases} \mathit{alua}, & \text{if } \mathit{borrow2} \wedge (\mathit{mul} = 3) \vee \neg \mathit{borrow2} \wedge (\mathit{mul} = 0), \\ \mathit{alua} + \mathit{alub}, & \text{if } \mathit{borrow2} \wedge (\mathit{mul} = 0) \vee (\mathit{mul} = 1), \\ \mathit{alua} - \mathit{alub}, & \text{otherwise.} \end{cases}$$

Here *alua* is the destination register, *alub* is the shifted multiplier, *mul* stores two bits of the multiplicand and *borrow2* is the borrow status.

As with the block data transfers, an invariant is needed to establish the state of the processor during each execute cycle. The final state of the destination register is shown to be the product of the register arguments.

2.5 Exceptions

The ARM6 verification is currently being extended to include resets, memory aborts and interrupts (both fast and normal). These exceptions are triggered by external signals and so the basic correctness model (Section 3.1) is no longer adequate. A correctness model with input streams [8] has been formalised in HOL – the definition of correctness is presented in Section 3.2.

At the time of writing, the ISA and ARM6 specifications have been extended to model exceptions. For example, the next-state function for the ARM6 now takes four additional values: `NRESET`, `ABORT`, `NFQ` and `NIQ`. Suitable data, stream and temporal abstractions have also been defined. Work is in progress on verifying correctness.

Note that the exceptions, and the associated abstractions, are being modelled deterministically. Throughout the project, the aim has been to ensure that the specifications are executable and that the abstraction mechanisms is made explicit.

3 Correctness

Section 3.1 defines correctness for two isolated systems at different levels of abstraction. This is then generalised in Section 3.2 to include input from the environment, modelled with input streams. The basic model has been used to verify the correctness of the entire ARM6 instruction set. With the inclusion of external exceptions, the model with input is now being used.

Correctness is defined with reference to all times at the abstract system level. For verification, it is shown that *under certain circumstances* it is possible to consider just one time step i.e. from time zero to one [9].

3.1 Basic Model

Definition 1 (Iterated map state functions). *Given a state space (non-empty set) A then state $: \mathbb{N} \to A \to A$ is an iterated map state function with initialisation function init $: A \to A$ and next state function next $: A \to A$ if, and only if*

$$\begin{aligned} state(0)(a) &= init(a), \\ state(t+1)(a) &= next(state(t)(a)). \end{aligned}$$

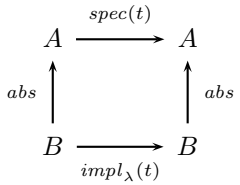
Definition 2 (Immersions). *A function $\lambda : A \to \mathbb{N} \to \mathbb{N}$ is an immersion if, and only if, for all $a \in A$, $\lambda(a)(0) = 0$ and for all $t_1 < t_2$, $\lambda(a)(t_1) < \lambda(a)(t_2)$.*

Definition 3 (Data abstractions). *A function $abs : B \to A$ is a data abstraction for initialisation functions $init_I : B \to B$ and $init_S : A \to A$ if, and only if, for all $b \in init_I(B)$, $abs(b) \in init_S(A)$ and for all $a \in init_S(A)$ there exists $b \in init_I(B)$ such that $abs(b) = a$; where $f(D) = \text{Range}(f) = \{f(x) : x \in D\}$.*

Definition 4 (Correctness). *A state function $impl : \mathbb{N} \to B \to B$ is a correct implementation of a state function $spec : \mathbb{N} \to A \to A$ with respect to an immersion $\lambda : B \to \mathbb{N} \to \mathbb{N}$ and a data abstraction $abs : B \to A$ for $impl(0)$ and $spec(0)$ if, and only if, for all $b \in B$ and $t \in \mathbb{N}$*

$$spec(t)(abs(b)) = abs(impl(\lambda(b)(t))(b)).$$

Correctness holds when the following diagram commutes:



where $impl_\lambda(t)(b) = impl(\lambda(b)(t))(b)$. Note that *spec* and *impl* need not necessarily be iterated maps.

3.2 Input Stream Model

Definition 5 (Iterated map state functions with input). *Given a stream space $S_B \subseteq \mathbb{N} \to B$ ($S_B \neq \emptyset$) then state : $\mathbb{N} \to A \times S_B \to A$ is an iterated map state function with initialisation function init : $A \to A$ and next state function next : $A \to B \to A$ if, and only if*

$$state(0)(a, s) = init(a),$$
$$state(t + 1)(a, s) = next(state(t)(a, s))(s(t)).$$

Definition 6 (Stream abstractions). *A function smpl : $A \times S_B \to S_C$ is a stream abstraction if, and only if, for all $a \in A$ and $s \in S_C$ there exists $s' \in S_B$ such that $s = smpl(a, s')$.*

Definition 7 (Correctness with input). *A State function impl : $\mathbb{N} \to C \times S_D \to C$ is a correct implementation of state function spec : $\mathbb{N} \to A \times S_B \to A$ with respect to an immersion $\lambda : C \times S_D \to \mathbb{N} \to \mathbb{N}$, data abstraction abs : $C \to A$ and stream abstraction smpl : $C \times S_D \to S_B$ if, and only if, for all $x \in C \times S_D$ and $t \in \mathbb{N}$*

$$spec(t)(abs \circ fst(x), smpl(x)) = abs(impl(\lambda(x)(t))(x)).$$

Correctness holds when the following diagram commutes:

$$\begin{array}{ccc} A \times S_B & \xrightarrow{spec(t)} & A \\ {\scriptstyle (abs \circ fst, smpl)} \uparrow & & \uparrow {\scriptstyle abs} \\ C \times S_D & \xrightarrow[impl_\lambda(t)]{} & C \end{array}$$

where $(f, g)(x) = (f(x), g(x))$.

4 Future Work

Future work will focus on producing more extensive models of ARM based systems. This will include looking at the co-processor and other ARM bus interfaces, such as AMBA. We will also aim to introduce higher levels of abstraction, so as to reason about small programs and investigate hardware-software co-design.

ARM processors are used to implement devices like mobile phones and PDAs, and so case studies will be developed with this in mind. In particular, modelling system-on-chip devices in which data security is important. Here, formal reasoning and correctness assurances are likely to add particular value. Examples may use a framework that is loosely based on ARM's TrustZone architecture:

> *A new Monitor mode within the core acts as a gatekeeper to identify secure code and reliably switch the system between secure and non-secure states. When the monitor switches the system to the secure state, the processor core gains additional levels of privilege to run trusted code, and to handle tasks such as authentication, signature manipulation and the processing of secure transactions.*
>
> `www.arm.com/products/CPUs/arch-trustzone.html`

References

1. M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In *CHARME 2001*, volume 2144 of *LNCS*, pages 433–448. Springer, 2001.
2. B. Barras. Programming and computing in HOL. In M. Aagaard and J. Harrison, editors, *TPHOLs 2000*, volume 1869 of *LNCS*, pages 17–37. Springer, 2000.
3. S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and T. Enrico, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2003.
4. B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. In M. K. Srivas and A. Camilleri, editors, *FMCAD '96*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996.
5. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Proceedings of the 6th International Conference, CAV '94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Berlin, 1994. Springer-Verlag.
6. A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.
7. D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, 1993.
8. A. Fox. *Algebraic Models for Advanced Microprocessors*. PhD thesis, University of Wales Swansea, 1998.
9. A. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge, Computer Laboratory, Apr. 2001.
10. A. Fox. A HOL specification of the ARM instruction set architecture. Technical Report 545, University of Cambridge Computer Laboratory, June 2001.
11. S. Furber. *ARM: system-on-chip architecture*. Addison-Wesley, second edition, 2000.
12. B. T. Graham. *The SECD Microprocessor, A Verification Case Study*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1992.
13. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2nd edition, 1996.
14. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213, 2003.
15. W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNCS*. Springer-Verlag, 1994.
16. R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In G. Gopalakrishnan and P. J. Windley, editors, *FMCAD 1998*, volume 1522 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
17. R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Formal verification of out-of-order execution with incremental flushing. *Formal Methods in System Design*, 20(2):139–158, Mar. 2002.

18. J. J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.
19. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
20. P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design, FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 2000.
21. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.
22. S. P. Miller and M. K. Srivas. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, Mar. 1996.
23. J. O'Leary, X. Zhao, R. Gerth, and C. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 3(1), 1999.
24. D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.
25. J. Sawada and W. A. Hunt, Jr. Verification of FM9801: An out-of-order model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, 20(2):187–222, Mar. 2002.
26. D. P. Schostak. *Methodology for the Formal Specification of RTL RISC Processor Designs (With Particular Reference to the ARM6)*. PhD thesis, The University of Leeds School of Computing, 2003.
27. D. Seal, editor. *ARM Architectural Reference Manual*. Addison-Wesley, second edition, 2000.
28. S. Tahar and R. Kumar. A practical methodology for the formal verification of RISC processors. *Formal Methods in System Design*, 13(2):159–225, Sept. 2002.
29. M. N. Velev. Formal verification of VLIW microprocessors with speculative execution. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 296–311, 2000.
30. P. J. Windley and M. L. Coe. A correctness model for pipelined microprocessors. In R. Kumar and T. Kropf, editors, *TPCD '94*, volume 901 of *LNCS*, pages 33–51. Springer-Verlag, 1995.
31. W. Wong. Modelling bit vectors in HOL: The word library. In J. J. Joyce and C.-J. H. Seger, editors, *HUG '93*, volume 780 of *LNCS*, pages 371–384. Springer-Verlag, 1994.

Building Extensible Compilers in a Formal Framework*

A Formal Framework User's Perspective

Nathaniel Gray, Jason Hickey, Aleksey Nogin, and Cristian Ţăpuş

Caltech, M/C 256-80
1200 E California Blvd
Pasadena, CA 91125, USA
{n8gray,jyh,nogin,crt}@cs.caltech.edu

Abstract. We outline a new methodology for compiler design, based on the use of a transformation logic defined within an existing general-purpose logical framework. We discuss how this methodology can be used to address several central issues in compiler design and implementation: ease of implementation, extensibility, compositionality, and trust. We show how pre-existing features of the logical framework we use help in compiler implementation; and we also discuss which features need to be added to the framework in order to facilitate our approach to compiler development.

1 Introduction

We are developing a new methodology for compiler design, based on the use of a transformation logic defined within an existing general-purpose logical framework. In our approach the central part of the compiler is a set of specifications on a formal language; these specifications follow a standard textbook account of programming language semantics almost to the letter. Most of the work required to turn these specifications into an actual compiler is handled *automatically* by the logical framework. We demonstrate how this methodology can be used to address several central issues in compiler design and implementation: ease of implementation, extensibility, compositionality, and trust.

We use the MetaPRL formal tool [9,11], which provides a well-defined syntax of terms, types, and programs. We represent programs and program transformations using higher-order abstract syntax (HOAS); binding, scoping, and substitution are handled automatically by the framework. The HOAS also allows mixing the object language (that contains operators like "`let`") with the meta-language (that contains operators like "`CPS`"), explicitly expressing the intermediate states of the compilation process. In addition, the framework provides a

* This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

rich tactic language for guiding proofs and transformations and for automatically extracting such guidance information from annotated specifications. Finally, the framework provides us with an interactive program refinement mode (initially designed for interactive formal proof development) and together with the explicit meta-language it proved to be an extremely powerful debugging tool.

Compositionality is a well-established principle in the construction of logical theories. In the compiler domain, we take a similar approach to compositionality and extensibility. The compiler defines a core theory for System F (variables, functions, application, and second order quantifiers) that is divided into transformation stages including type inference, type checking, CPS transformation, closure conversion, and assembly code generation. Additional components for Boolean values, arithmetic, tuples, arrays, recursive functions, etc., are defined as independent extensions. Each extension defines its own set of formal rules for each transformation stage and adds new strategy code to the tactic used to control that stage. By locally ensuring that the component acts as a conservative extension of the core and other components it is derived from, we get a strong guarantee that there will be no unexpected interactions between different compiler modules or different language features.

Another extremely important and challenging issue in compiler development is reliability and trust. In the context of a compiler, it is useful to focus on the code where flaws have the potential to cause the compiler to produce incorrect output for some input program—we call such code *trusted*. Flaws in *untrusted* code may cause the compiler to fail to produce output on some valid input programs, but they cannot cause the compiler to produce incorrect output.

When a compiler is implemented in a general purpose language, it is often difficult to isolate the parts of the compiler that must be trusted, and in the worst case the entire code base must be trusted. Trust is also a central issue in compositionality and ease of implementation. If the invariants that specify the compiler involve complex interactions between many parts of the implementation, maintaining and extending the compiler can be quite difficult.

In our approach, the compiler is built in the style of the LCF theorem prover [4]. The program transformations are each defined in two parts: a set of trusted transformation *axioms* and untrusted *tactic code* to direct the transformation strategy. The transformation axioms are defined in a formal logic using notation similar to that in the literature, they represent only a small part of the compiler, and they are *verifiable*. That is, the entire trusted code path is small, precisely and formally defined, and it may be validated against a program semantics if desired. Note, however, that we do not consider verifiability to be the primary concern of this work. We believe that there is substantial value in significantly reducing the amount of trusted compiler code, even if it is not completely eliminated.

A number of guarantees are provided by the framework itself. For example, the HOAS implementation ensures that program transformations are never allowed to violate scoping or accidentally capture a variable. Even the framework implementation does not have to be trusted—the tool is capable of retaining

and providing a full log of the program transformations performed during the compilation process; if an extreme level of confidence is needed, an independent checker could be implemented.

1.1 Overview

This paper is based on a case study of a working compiler implementation for an ML-like source language [7], compiled to assembly code for the Intel x86 machine architecture. As mentioned, the core is based on the language of System F. There are extensions for 1) additional base types like Boolean values and integers, 2) aggregates like arrays and tuples, and 3) recursive functions. The backend uses HOAS to define a scoped x86 assembly language [7,10]. The compiler stages include type inference, type checking, CPS transformation, closure conversion, and assembly code generation. The compiler is implemented in the MetaPRL logical framework.

This paper focuses on demonstrating how the features of the logical framework help to implement the compiler and improve its trustworthiness and extensibility. In our implementation we were able to precisely and concisely define each of the standard compiler stages (excluding parsing and pretty-printing of the output assembly) formally. The precision comes from using the formal notation, and the brevity follows from the rich set of tools provided by the logical framework. We begin the account with a description of terminology (Section 2) and the overall compiler architecture (Section 3), and follow it with a description of a few of the key stages of the compiler. As a demonstration of our approach, we present the CPS stage of the compiler (Section 4) based on the work of Danvy and Filinski [3] and show how the use of HOAS and derived rules in logical framework can make our *implementation* simpler that Danvy and Filinski's original account. Finally, Section 5 provides a discussion of our experiences and give some ideas for further improvements of the methodology and Section 6 discusses related work.

2 MetaPRL

All logical syntax in the MetaPRL framework is expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name (like "sum"), which is a unique name identifying the kind of term; 2) a list of parameters representing constant values; and 3) a set of subterms with possible variable bindings. We use the following syntax to describe terms:

$$\underbrace{opname}_{operator\ name}\ \underbrace{[p_1;\cdots;p_n]}_{parameters}\underbrace{\{\vec{v}_1.t_1;\cdots;\vec{v}_m.t_m\}}_{subterms}$$

All the free occurrences of variables $\vec{v}_i$ in t_i will be considered bound by the operator. When $n = 0$, the parameter brackets are omitted; when $\vec{v}_i$ is empty, the dot before t_i is usually omitted.

Below are a few examples of terms that could be used in a formalization of a simple lambda calculus.

Pretty-printed form	Term
1	`integer[1]{}`
$\lambda x.b$	`lambda[]{ x. b }`
$f(a)$	`apply[] { f; a }`
$x + y$	`sum[]{ x; y }`

Numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Each operator has a fixed arity, which includes a fixed number of parameters, a fixed number of subterms and a fixed number of bindings for each subterm. (More specifically, if two operators have different arities, they will be considered to be distinct even if they happen to have the same opname.)

In addition to the basic term language described above, the framework also provides three special kinds of terms. The first one is the simple first-order (object language) variables. These are the variables that can be bound in a term.

Another class of special terms are second-order (meta-level) variables, which are patterns used to define scoping and substitution [16]. A second-order variable pattern has the form $V[v_1; \cdots ; v_n]$, which represents an arbitrary term that may have free first-order variables $v_1, \ldots, v_n$. The corresponding substitution has the form $V[t_1; \cdots ; t_n]$, which specifies the simultaneous, capture-avoiding substitution of terms $t_1, \ldots, t_n$ for $v_1, \ldots, v_n$ in the term matched by V. Second-order variables are used to specify logical rules and term rewrites.

A term rewrite states that any term that matches the left-hand-side of the rewrite (its *redex*) can be replaced with the corresponding value of the right-hand-side of the rewrite (its *contractum*), and vice-versa, in any context. For example, β-reduction could be specified with the following rewrite.

$$(\lambda x.v_1[x])\ v_2 \leftarrow [\text{beta}] \rightarrow v_1[v_2]$$

The $v_1[x]$ in the redex stands for an arbitrary term that may have free occurrences of the first-order variable x, and v_2 is another arbitrary term. The meta-term $v_1[v_2]$ in the contractum specifies the substitution of the term matched by v_2 for x in v_1.

Second-order notation can also express the *lack* of bound occurrences of a certain variable. The following rewrite is valid in second-order notation and would be provable in the presence of the β-reduction rewrite.

$$(\lambda x.v[])\ 1 \leftarrow [\text{const}] \rightarrow (\lambda x.v[])\ 2$$

In the context λx, the second-order variable $v[]$ matches only those terms that do not have x as a free variable. No substitution is performed; the β-reduction of both sides of the rewrite yields $v[] \longleftrightarrow v[]$, which is valid reflexively. Normally, when a second-order variable $v[]$ has an empty argument list $[]$, we omit the brackets and use the simpler notation v.

The last class of special terms is sequents (sometime also called telescope terms) of the form

$$x_1 : t_1; \ldots; x_n : t_n \vdash_a c,$$

where n can be 0. The term c is the *conclusion* of the sequent; the terms t_i are its *hypotheses*; the variables x_i introduce binding occurrences (each x_i is bound in all t_j for $j > i$ and in c). Finally, the term a is the *sequent argument* that specifies what kind of sequent it is—essentially the argument plays the same role for sequents as the operator name plays for ordinary terms. Sequent *schemas* [16] may also include *context* meta-level variables that stand for arbitrary lists of hypotheses. For example, the sequent schema

$$\Gamma;\, x : T[];\, \Delta[x] \vdash_{a[]} c[x]$$

(where Γ and Δ are context variables and T, a and c are second-order variables) stands for an arbitrary sequent with at least one hypothesis.

The compilation process is expressed in MetaPRL as a judgment of the form $\Gamma \vdash \langle\langle e \rangle\rangle$, which states that the program e is compilable in the logical context Γ. The exact meaning of the $\langle\langle e \rangle\rangle$ judgment is defined by the target architecture. A program e' is compilable if it can be represented by a sequence of valid assembly instructions. The compilation task is a process of rewriting the source program e to an equivalent assembly program e'.

MetaPRL uses OCaml [19] as its tactic construction language in the LCF style. When an inference rule or a rewrite rule is defined in MetaPRL, the framework creates an OCaml expression that can be used to apply the rule. Code to guide the application of rules and rewrites is written in OCaml, using a rich set of primitives provided by MetaPRL. In addition, MetaPRL automates the construction of most guidance code.

3 Compiler Overview

A compiler is defined by a sequence of transformations that take a program in a source language and translate it to a program in a target language. In this case study, the full source language is an ML-like source language with type inference and higher-order functions and the target language is the x86 assembly language.

Figure 1 shows a diagram of the compiler architecture, where the core and the extensions are represented horizontally. Extensions do not have to define code for each of the stages; for example, closure conversion applies only to functions, and the other extensions may ignore it. Extensions may also have dependencies upon one another, as shown by the arrows on the left of each extension: tuples require integers, which require general operations for arithmetic, which require Boolean values for relations.

The compiler includes an initial informal phase that uses the Phobos extensible parser to convert the textual source code to the term representation used by the logical framework [5].

The syntax for the typed intermediate language for the case study is shown in Figure 2. The source language is similar, except it is untyped. For clarity, the

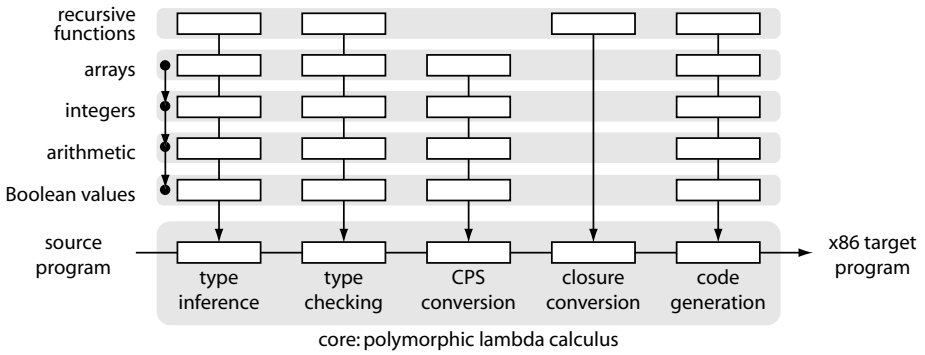


Fig. 1. The high-level compiler architecture is designed around a sequence of transformations for a core language based on the polymorphic lambda calculus. Each extension defines new types and values, as well as an extension to each of the core stages. The vertical arrows indicate extensions to core stages; the code is structured horizontally.

syntax is shown in the pretty-printed form; internally each of the expressions and types uses native MetaPRL notation.

The arities of functions, application, type abstractions, type applications, and tuples are unconstrained. Internally, functions and their types use sequent notation. For example, the sequent $x_1 : t_1, \ldots, x_n : t_n \vdash_\kappa e$ is used to represent the function $\lambda_\kappa(x_1 : t_1, \ldots, x_n : t_n).e$. There are three kinds of functions and application: λ_r represents a recursive function (f is the recursive binding); λ_s represents a "normal" function; an application $e(e_1, \ldots, e_n : t_1, \ldots, t_n)_c$ represents a closure (the runtime passes the arguments as a tuple).

4 Example: CPS Conversion

The implementation of CPS conversion is a good illustration of our methodology. We wish to demonstrate both that 1) the formal definition of the compiler transformations is natural, and 2) that the methodology is compositional. We present a very straightforward implementation based on the ability of the framework to combine the meta-language and the object language and we will show how the tail recursive optimizations can be derived formally from the eta reduction.

We use a higher-order variant of Danvy and Filinski's approach to CPS conversion [3]. We start by adding a new term to the meta-language—$\texttt{CPS}\{e; t; v.c[v]\}$, where the first argument e is the expression that is being converted, the second argument t is the type of that expression and the third argument is the *meta-continuation* of the CPS process. In other words, c is the *rest* of the program and v marks the location where the CPS of e should go.

The following rule specifies CPS for variables of the object language.

$$\texttt{CPS}\{!x; t; v.c[v]\} \leftarrow [\text{cps\_var}] \rightarrow c[!x]$$

Expressions		Types	
Core language			
$e ::= x$	Variables	$t ::= \alpha$	Variables
$\mid (e : t)$	Type constraint	$\mid \bot$	Empty type
$\mid \mathbf{let}\, v : t\ =\ e_1\ \mathbf{in}\ e_2$		$\mid \top$	All programs
$\mid \lambda_\kappa(x_1 : t_1, \ldots, x_n : t_n).e$	Functions	$\mid (t_1, \ldots, t_n) \rightarrow_\kappa t$	Function types
$\mid e(e_1, \ldots, e_n : t_1, \ldots, t_n)_\kappa$	Application		
$\mid \Lambda(\alpha_1, \ldots, \alpha_n).t$	Type abstraction	$\mid \forall(\alpha_1, \ldots, \alpha_n).t$	Polymorphism
$\mid e[t_1, \ldots, t_n]$	Type application		
Boolean values			
$\mid$ *true* $\mid$ *false*	Constants	$\mid \mathbb{B}$	Boolean type
$\mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e$	Conditional		
Integers			
$\mid i$	Constants	$\mid \mathbb{Z}$	Integer type
$\mid e\ \mathit{binop}\ e$	Arithmetic		
$\mid e\ \mathit{relop}\ e$	Relations		
Tuples			
$\mid (e_1, \ldots, e_n)$	Tuples	$\mid t_1 * \cdots * t_n$	Product type
$\mid (e : t).i$	Projection		

Recursive functions

$\lambda_r(x_1 : t_1, \ldots, x_n : t_n, f : t).e$

Function kinds

$\kappa ::= s \mid c \mid r$

$\mathit{binop} ::= + \mid - \mid \cdots$ Binary operations

$\mathit{relop} ::= < \mid \leq \mid \cdots$ Binary relations

Fig. 2. The typed intermediate language is based on the polymorphic lambda calculus. Extensions add Boolean values, arithmetic, tuples, arrays (not shown), and recursive functions. The source language is a type erased version of the intermediate language.

The notation $!x$ is MetaPRL syntax for first-order variables that are bound outside of the local scope of the rewrite rule. In this rule, the meta-continuation is consumed. The rewrite puts the variable into the appropriate location and returns the whole expression. Note that we use meta-language notation in place of Danvy and Filinski's "static" operators $\overline{@}$ and $\overline{\lambda}$.

In the rule for **let** expressions, a new meta-continuation is created.

$$
\begin{array}{l}
\mathtt{CPS}\{\mathbf{let}\, v_1 : t_1 = e_1\ \mathbf{in}\ e_2[v_1]; t_2; v_2.c[v_2]\} \\
\quad \leftarrow [\mathrm{cps\_let}] \rightarrow \\
\mathtt{CPS}\{e_1; t_1; v_3.\mathbf{let}\, v_1 : \mathtt{TyCPS}\{t_1\} = v_3\ \mathbf{in} \\
\qquad\qquad \mathtt{CPS}\{e_2[v_1]; t_2; v_2.c[v_2]\}\,\}
\end{array}
$$

`TyCPS` here is a meta-term that is used to specify the CPS conversion for types (adding an extra argument to all function types) similarly to how the `CPS` term is used to specify the CPS conversion for expressions.

The rule for the CPS of applications could be specified the following way:

$$\begin{array}{l}\texttt{CPS}\{f(es : ts); t; v.c[v]\}\\ \quad \leftarrow [\text{cps\_apply}] \rightarrow\\ \texttt{CPS}\{f; ts \rightarrow t; v_f.\\ \texttt{CPS}\{es; ts; v_e.\\ \textbf{let}\, c_2 : (\texttt{TyCPS}\{t\} \rightarrow \bot) = \lambda_s v : \texttt{TyCPS}\{t\}.c[v]\ \textbf{in}\\ v_f(c_2, v_e : (\texttt{TyCPS}\{t\} \rightarrow \bot), \texttt{TyCPS}\{ts\})\}\}\end{array}$$

where *es* and *ts* are second-order variables used to match *lists* of arguments and types respectively.

In our implementation we add a meta-let operation to the meta-language.

$$\textbf{meta\_let}\, v = e_1\ \textbf{in}\ e_2[v] \leftarrow [\text{meta\_let}] \rightarrow e_2[e_1]$$

Using this operation, the `cps_apply` rule is written as follows.

$$\begin{array}{l}\texttt{CPS}\{f(es : ts); t; v.c[v]\}\\ \quad \leftarrow [\text{cps\_apply}] \rightarrow\\ \texttt{CPS}\{f; ts \rightarrow t; v_f.\\ \texttt{CPS}\{es; ts; v_e.\\ \textbf{meta\_let}\, t' = \texttt{TyCPS}\{t\}\ \textbf{in}\\ \textbf{meta\_let}\, t'' = t' \rightarrow \bot\ \textbf{in}\\ \textbf{let}\, c_2 : t'' = \lambda_s v : t'.c[v]\ \textbf{in}\\ v_f(c_2, v_e : t'', \texttt{TyCPS}\{ts\})\}\}\end{array}$$

This is more efficient as the type t will only have to be converted once, not 3 times. Again, the ability to combine the object language with meta-language yields very compact straightforward and precise formal code.

The ability to manipulate the meta-continuations also helps making the rules for the conversion of the argument lists very concise.

$$\begin{array}{l}\texttt{CPS}\{e_1 :: es; t_1 :: ts; v.c[v]\}\\ \quad \leftarrow [\text{cps\_args\_cons}] \rightarrow\\ \texttt{CPS}\{e_1; t_1; v_1.\texttt{CPS}\{es; ts; vs.c[v_1 :: vs]\}\}\end{array}$$

$$\texttt{CPS}\{(); (); v.c[v]\} \leftarrow [\text{cps\_args\_nil}] \rightarrow c[()]$$

Below is an example of a CPS rewrite from the Boolean extension, written in the MetaPRL native syntax.

```
prim_rw cps_true {| cps |} :
  CPS{bTrue; TyBool; v. 'c['v]}
  <-->
  'c[bTrue]
```

The above 4 lines are the only code that needs to be added to the system for it to know how to handle the `true` constant in the CPS stage. The system does not require this code to go in a specific place. The `{| cps |}` annotation specifies that this rewrite should be added to the lookup table [8] used by the CPS tactic.

In addition to the basic CPS transformation, we define a tail-recursive version as $\texttt{TailCPS}\{e; t; k\} := \texttt{CPS}\{e; t; v.k(v)\}$. Using this definition we *formally derive* the tail call optimizations using the eta reduction rule.

5 Conclusions and Future Work

During the course of this work on the case study, we found that the implementation was easier than we expected, in part because the ability to mix the object and meta-language freely gave us more power than we anticipated. Because the account mirrors standard semantics textbook specifications very closely and the amount of code that must be trusted is only a few hundred lines, it is relatively easy to believe in its correctness. The mechanisms for extensions and compositionality provided by the logical framework generalized naturally to the compiler design.

On the compiler structure side, there are many open avenues to explore. We plan to investigate bounded polymorphism, which we will use for object systems and extensible tuples. The current core language already provides preliminary, but incomplete support. We also plan to develop a representation of mutually recursive functions, which will require extending the support provided by the logical framework.

One apparent challenge of our approach is that all program transformations must be constructed from a fixed number of rewrite rules that each describe a pattern over a fixed number of program points. In other words, global program transformations must be composed of a sequence of local transformations, and it is not always obvious how to do this. In addition, global transformations may require knowledge of the entire program syntax, which can be at odds with compositionality. In our experience, however, we have found this problem to be much easier to solve than we originally expected; all of the transformations we have implemented so far have been easy to break into appropriately localized pieces. On the other hand, we have not yet tried formalizing optimization techniques that are normally implemented using global program analysis, such as global code motion; the problem of breaking these types of transformations into localized rewrites could be harder.

For the most part, our work concentrated on implementing the compiler without modifying the existing logical framework. However in the future we are likely to try adding some additional features to the framework to facilitate compiler implementation. There are two main limitations that we are planning to address—recursive variable-arity binding structure and context-aware rewriting.

5.1 Recursive Binding Structure

Recursive functions are a very basic feature of ML-like languages. In general, recursive functions have the following form.

$$\begin{array}{l}\texttt{let rec } f_1\ x_1\ \ldots\ x_{k_1} = e_1\\ \quad\texttt{and } f_2\ x_1\ \ldots\ x_{k_2} = e_2\\ \qquad\vdots\\ \quad\texttt{and } f_n\ x_1\ \ldots\ x_{k_n} = e_n\\ \quad\texttt{in } \ e\end{array}$$

There are two difficulties associated with the above—first, the functions have variable arity, and second, the functions are mutually recursive and each of the e_i may have free occurrences of each of the f_j.

As we describe in Section 3, variable arity functions could be implemented by using a sequent representation. Mutual recursion is more challenging. One approach would be to pack the mutually recursive functions into a record and then define the record recursively [10]. Defining a *single* variable recursively is easy, but in this approach function variables turn into explicit record field *names* and are no longer mapped to normal variables. As a result, most of the advantages provided by HOAS are lost and the labels have to be managed (and alpha-renamed) explicitly.

A proper HOAS solution would be to introduce a new kind of sequent to the logical framework—a *recursive sequent* of the form

$$x_1 : t_1 = e_1;\ \ldots;\ x_n : t_n = e_n \vdash e$$

where each x_i is bound in all the subsequent t_j $(j > i)$, in *all* of the e_k $(1 \le k \le n)$, and in e. The traditional sequent mechanism can be subsumed by recursive sequents by making the e_i optional.

5.2 Context-Aware and Conditional Rewriting

Consider the following trivial optimization rewrite:

$$\textbf{let}\, v : t = e\ \textbf{in}\ v \leftarrow [\text{let\_opt}] \rightarrow e$$

Depending on the exact semantics used, this rewrite could be considered invalid since it potentially allows turning mistyped expressions into well-typed ones and vice-versa (remember that rewrites are bidirectional). In this simple example, the rewrite can be fixed relatively easily by adding an explicit type constraint to the contractum as follows.

$$\textbf{let}\, v : t = e\ \textbf{in}\ v \leftarrow [\text{let\_opt}] \rightarrow e : t$$

However, we would generally like to be able to express rewrites that are only conditionally applicable. In particular, we would like to specify conditions of the forms "*applicable in a context that expects the redex to have type t*" and "*applicable when subterm e is well-typed.*" While the MetaPRL system does provide support for conditional rewriting, not all conditions that are natural in the compiler implementation domain are easily expressible in MetaPRL.

6 Related Work

`FreshML` [17] adds to the ML language support for straightforward encoding of variable bindings and alpha-equivalence classes. Our approach differs in several important ways. Substitution and testing for free occurrences of variables are explicit operations in `FreshML`, while MetaPRL provides a convenient implicit syntax for these operations. Binding names in `FreshML` are inaccessible, while only the formal parts of MetaPRL are prohibited from accessing the names. Informal portions—such as code to print debugging messages to the compiler writer, or warning and error messages to the compiler user—can access the binding names, which aids development and debugging. `FreshML` is primarily an effort to add automation; it does not address the issue of validation directly.

Liang [13] implemented a compiler for a simple imperative language using a higher-order abstract syntax implementation in λProlog. Liang's approach includes several of the phases we describe here, including parsing, CPS conversion, and code generation using a instruction set defined using higher-abstract syntax (although in Liang's case, registers are referred to indirectly through a meta-level store, and we represent registers directly as variables). Liang does not address the issue of validation in this work, and the primary role of λProlog is to simplify the compiler implementation. In contrast to our approach, in Liang's work the entire compiler was implemented in λProlog, even the parts of the compiler where implementation in a more traditional language might have been more convenient (such as register allocation code).

Hannan and Pfenning [6] constructed a verified compiler in LF (as realized in the Elf programming language) for the untyped lambda calculus and a variant of the CAM [2] runtime. This work formalizes both compiler transformation and verifications as deductive systems, and verification is against an operational semantics.

Previous work has also focused on augmenting compilers with formal tools. Instead of trying to split the compiler into a formal part and a heuristic part, one can attempt to treat the *whole* compiler as a heuristic adding some external code that would watch over what the compiler is doing and try to establish the equivalence of the intermediate and final results. For example, the work of Necula and Lee [14,15] has led to effective mechanisms for certifying the output of compilers (e.g., with respect to type and memory-access safety), and for verifying that intermediate transformations on the code preserve its semantics. Pnueli, Siegel, and Singerman [18] perform verification in a similar way, not by validating the compiler, but by validating the result of a transformation using simulation-based reasoning.

Semantics-directed compilation [12] is aimed at allowing language designers to generate compilers from high-level semantic specifications. Although it has some overlap with our work, it does not address the issue of trust in the compiler. No proof is generated to accompany the compiler, and the compiler generator must be trusted if the generated compiler is to be trusted.

Boyle, Resler, and Winter [1], outline an approach to building trusted compilers that is similar to our own. Like us, they propose using rewrites to transform

code during compilation. Winter develops this further in the HATS system [20] with a special-purpose transformation grammar. An advantage of this approach is that the transformation language can be tailored for the compilation process. However, this significantly restricts the generality of the approach, and limits re-use of existing methods and tools.

References

1. J. Boyle, R. Resler, and K. Winter. Do you trust your compiler? Applying formal methods to constructing high-assurance compilers. In *High-Assurance Systems Engineering Workshop*, Washington, DC, August 1997.
2. G. Cousineau, P.L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.
3. Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
4. Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.
5. Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences*. IEEE, 2002.
6. John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings of the 7th Symposium on Logic in Computer Science*. IEEE, IEEE Computer Society Press, 1992.
7. Jason Hickey, Nathan Gray, Aleksey Nogin, and Cristian Tapus. Reliable frameworks for extensible compilers. In preparation, 2004.
8. Jason Hickey and Aleksey Nogin. Extensible hierarchical tactic construction in a logical framework. Accepted to the TPHOLs 2004 conference, 2004.
9. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.
10. Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir. Compiler implementation in a formal logical framework. In *Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pages 1–13. ACM Press, 2003. `http://doi.acm.org/10.1145/976571.976575`. Extended version of the paper is available as Caltech Technical Report caltechCSTR:2003.002.
11. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
12. Peter Lee. *Realistic compiler generation*. MIT Press, 1989.
13. Chuck C. Liang. Compiler construction in higher order logic programming. In *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 47–63, 2002.
14. George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

15. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
16. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.
17. Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.
18. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.
19. Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, Paris, 2nd edition, 1999. In French.
20. Victor L. Winter. Program transformation in hats. In *Proceedings of the Software Transformation Systems Workshop*, May 1999.

Compiling HOL4 to Native Code

Joe Hurd*

Computing Laboratory
Oxford University
joe.hurd@comlab.ox.ac.uk

Abstract. We present a framework for extracting and compiling proof tools and theories from a higher order logic theorem prover, so that the theorem prover can be used as a platform for supporting reasoning in other applications. The framework is demonstrated on a small application that uses HOL4 to find proofs of arbitrary first order logic formulas.

1 Introduction

The normal mode of use of a theorem prover is for the user to enter into a dialogue with the system, guiding the way towards the proof of one or more key theorems. The end result is a mechanically checked theory, which might demonstrate that a program meets its specification, say, or that a purported mathematical proof is in fact a valid argument.

A feature of theorem provers in the LCF tradition is that they provide a full programming language, ML, for implementing proof tools: programs that use the infrastructure of the theorem prover to ensure sound reasoning. The most common kind of proof tool is the ad-hoc tactic, implemented by the user to speed up the development of a mechanically checked theory. However, there is no reason why a proof tool should not be an arbitrary ML program that happens to use the theorem prover as a 'reasoning library'. In this alternative mode of use of the theorem prover, this program is the end-product, not a mechanically checked theory.

Motivating this work are two recent proof tools implemented using the HOL4 theorem prover[1] [3]:

- With the present author, Gordon [4] has built on a formalization of the temporal logic PSL by implementing a prototype proof tool. It takes as input a PSL formula, deduces an equivalent finite state automaton, and prints the finite state automaton in the form of a Verilog monitor that can be simulated with a circuit to check the property is never violated.
- Using their HOL4 specification of the TCP internet protocol, Bishop et al. [2] have implemented a proof tool that seeks to validate a trace of packets captured from a test network. Discrepancies between the specification of TCP and the implementation on the test network manifest themselves as failures to prove that a trace is legal.

* Supported by a Junior Research Fellowship at Magdalen College, Oxford.

[1] HOL4 is available at http://hol.sourceforge.net/.

Both the above proof tools are computationally expensive, and require no interaction with the user after the initial invocation. As such, they are prime candidates to optimize for speed. In this paper we describe the experience of porting HOL4 to a modern optimizing compiler, to make proof tools like these run as efficiently as possible. We present the general framework for compiling the theorem prover infrastructure, and then demonstrate it on a case study with the HOL4 first order prover.

The rest of the paper is structured as follows: section 2 describes the experience of porting HOL4 to the compiler; section 3 presents the results of a small experiment to show what efficiency gains are possible; and section 4 concludes.

2 Compiling HOL4 to Native Code

2.1 Assembling the Program Source Code

The source language for the HOL4 theorem prover is Standard ML, interpreted using Moscow ML.[2] The current distribution ships with 384 ML modules: 6 are simple utility functions; 20 form the logical kernel of the theorem prover; and 30 comprise the parser. Of the remaining 328 modules, 247 form a collection of proof tools (e.g., a simplifier) provided by the system, and 81 are mechanized theories (e.g., the real numbers) providing useful types, constants and lemmas that users might need.

The 81 mechanized theory files (`xTheory.sml`) contain theorem statements, but not any proofs. Users create theory files by executing a separate ML program called the proof script (`xScript.sml`): this calls the necessary proof tools to create all the theorems, which are then written out to the theory file. In later sessions, when `xTheory` is required, only the theory file needs to be loaded, the proofs do not need to be rechecked by the system every time. After downloading HOL4, the initial step is to build all of the theory files from the proof script files, after that all the theories that are part of the distribution are ready to be used.

The program that we wish to compile may both make calls to HOL4 proof tools and refer to the contents of mechanized theories. For example, the tool mentioned in the introduction for checking TCP traces makes use of several theory files in which the TCP protocol is modelled using operational semantics. Certainly we do not want all the proofs to be re-checked every time the program is invoked, and so we drop the proof script files, including only the generated theory files needed by the program.

The assembled program source code thus consists of:

1. simple utility functions;
2. the logical kernel;
3. the parser (needed by the generated theory files);
4. any proof tools directly called by the program;
5. any theory files used by the program;
6. and finally the program source code.

[2] Moscow ML is available at `http://www.dina.dk/~sestoft/mosml.html`.

2.2 Porting HOL4 to the MLton Compiler

Whereas Moscow ML translates ML source code into byte code which is then interpreted, the MLton[3] compiler translates ML source code directly into native code. In addition, MLton is a whole program compiler, so that functors and polymorphism can be eliminated to produce still more efficient code. Performance on various benchmarks[4] indicate that MLton produces the fastest running code of the leading Standard ML compilers, making it suitable to compile computationally expensive proof tools.

Although HOL4 is written in Standard ML, the source language for both Moscow ML and MLton, there are enough differences between the platforms to make porting non-trivial. For example, in several primitive inference rules, a check must be performed to see whether two lambda terms are α-equivalent. This check can be made more efficient using a pointer equality test, but this is not part of Standard ML. Both Moscow ML and MLton provide such a test, but differently in the two platforms.

Also, the Standard ML basis library is evolving at present, and there are two versions in current use: the 1997 version and the 2002 version. Moscow ML implements the 1997 version, plus some useful modules that are not part of the official basis library. MLton implements both versions, the user selects which one to use with a command-line argument. This part of the port was therefore easy: the 1997 basis was selected in MLton, and the extra modules in Moscow ML were ported to MLton.

The hardest part of the port was the HOL4 lexer. As part of the distribution, Moscow ML provides an efficient lexer generator called `mosmllex`, and this is used to generate the HOL4 lexer. Unfortunately, despite being type safe, the code generated by `mosmllex` does not pass the Standard ML type checker, and Moscow ML casting operations are deployed in the generated code to avoid type clashes. Creating an equivalent lexer in Standard ML required manually altering the HOL4 lexer to use a suitable union type that included all types that caused a clash.

Finally, and most seriously, there were problems associated with the size of the assembled source code. The size of the program for validating TCP traces mentioned in the introduction comes to 440,000 lines of Standard ML. This breaks down as 170,000 lines for files in the HOL4 distribution, and 270,000 lines for theories and tools in the trace checker itself. Most of the bulk is a result of large HOL4 datatype declarations, which automatically generate theorems about induction, cases and representation. Despite dense packing in the theory files by making use of term/type sharing and abbreviations, five theory files in the trace checker are each more than 10,000 lines long. At the time of writing, MLton has performance problems beyond about 150,000 lines of source code, and so we have not been able to test the TCP trace checker.[5] Instead, we restrict ourselves

[3] MLton is available at `http://www.mlton.org/`.

[4] Data from `http://www.mlton.org/performance.html`.

[5] However, the MLton team are actively working on improvements that will permit the compilation of such large programs.

to compiling applications that use a subset of HOL4, such as the following case study.

3 Case Study: First Order Proof

Provided with the HOL4 theorem prover is a proof tactic called `METIS_TAC` that uses ordered resolution to search for a first order refutation on the input goal, and if successful translates the proof to higher order logic [5]. This tactic has evolved somewhat since its initial deployment, and amongst other improvements now converts formulas to clauses using *definitional CNF*, where new variables are introduced successively (in a greedy fashion) to minimize the number of clauses.

In this experiment, we create a HOL4 proof tool that reads in a first order formula in TPTP[6] syntax and sets it as a proof goal, and then tries to prove it by invoking the `METIS_TAC` tactic. The advantage of such an experiment is that it gives us two points of comparison: firstly the MLton compiled version of the proof tool can be compared to the Moscow ML interpreted version; and secondly both can be compared to the results of other first order provers on the same problems.

This experiment made use of a RedHat 9 Linux box with a Pentium 4 3GHz processor and 4Gb of main memory (essential for compiling large programs with MLton), Moscow ML version 2.00, and MLton version 20040227. The problems all come from version 2.6.0 of the TPTP library.

Assembling the source code for the proof tool results in 60,000 lines of Standard ML, which includes three theory files used by the first order prover: booleans, combinatory logic and normal forms (such as CNF).[7] Compiling using MLton results in a 14Mb standalone executable, whereas doing the same in Moscow ML (using the `-standalone` compiler flag) results in a relatively small 0.5Mb executable.

We first look at problem `SYN007+1` in the TPTP library, which has the form

$$p_1 \iff (p_2 \iff (\cdots \iff (p_n \iff (p_1 \iff (p_2 \iff (\cdots \iff p_n)) \cdots)$$

where n is a problem parameter. When n is set to 14, the compiled version of `METIS_TAC` proves the goal in 4.5s. This makes use of HOL4 stripping tactics that reduce a goal of the form $P \iff Q$ to the two subgoals $P \Rightarrow Q$ and $Q \Rightarrow P$. Also, for each of the subgoals generated, the definitional CNF engine kicks in, and for the most extreme subgoal reduces the number of final clauses from 67,000,000 to a mere 100. Running the Moscow ML version of exactly the same program takes 63.5s.

We next run both versions of the prover on the same 70 first order formulas that were used in the 2003 CADE Automatic Theorem Prover System Competition[8]. To aid comparison with other provers' results in the competition, we set

[6] The TPTP problem library is available at `http://www.tptp.org/`.

[7] The largest version of HOL4 that was successfully compiled was a 120,000 line `METIS_TAC` self-test that used 26 theories.

[8] The CASC 2003 homepage is at `http://www.cs.miami.edu/~tptp/CASC/19/`.

the same time limit of 600s per problem, although it should be noted that we are running on a much faster machine with more memory than those used in the competition.

The MLton version of the prover solves 28 problems out of 70, which puts it between the 4th prover (DCTP 10.2p, at 42 problems) and the 5th prover (Otter 3.2, at 14 problems) out of the 6 that entered the first order formula division. For comparison, the top prover in this division was Vampire 5.0, which solved 57 out of 70 problems. The Moscow ML version of the prover solves 25 problems out of 70, missing 3 of the harder problems, which puts it at the same place in the results table. A graph showing the times that each version found proofs is shown in Fig. 1.

To calculate the average speed-up, we look at the 25 problems that both provers succeeded with, and calculate the geometric mean of the ratio between the times. This gives a speed-up factor of 10.3, which correlates with the present author's experiences porting other programs from Moscow ML to MLton.

4 Conclusions and Related Work

In this paper we have presented a framework for extracting theories and proof tools from a higher order logic theorem prover, and compiling them to native code using a modern optimizing compiler. This is a useful step along the road of embedding theorem proving inside other applications, such as compilers or question answering systems.

Although it is not yet possible to compile the proof tools that directly motivate this work, we are confident that further work on both the compiler and theorem prover sides will soon allow this to take place, bestowing a factor of 10 speed-up to the users with no change in functionality.

We have also seen that a simple wrapper allows HOL4 to compete with the first order provers in the CASC competition. No tuning of parameters took place before running the experiment: exactly the same proof tactic was used that is available to users during interactive proof.

The Twelf theorem prover[9] [6] has been ported to MLton, and provides an interface via a Twelf Standard ML module. Our work shares a similar approach of theorem prover as platform: in the case of Twelf a major application is proof-carrying code; we aim to support reasoning applications (such as those mentioned in the introduction) where higher order logic is a more convenient modelling language.

The Coq theorem prover[10] [1] has also been compiled to native code using the OCaml compiler, though the objective seems to be more speeding up the type checking of theories rather than providing a theorem prover platform.

[9] Twelf is available at `http://www-2.cs.cmu.edu/~twelf/`.
[10] Coq is available at `http://coq.inria.fr/`.

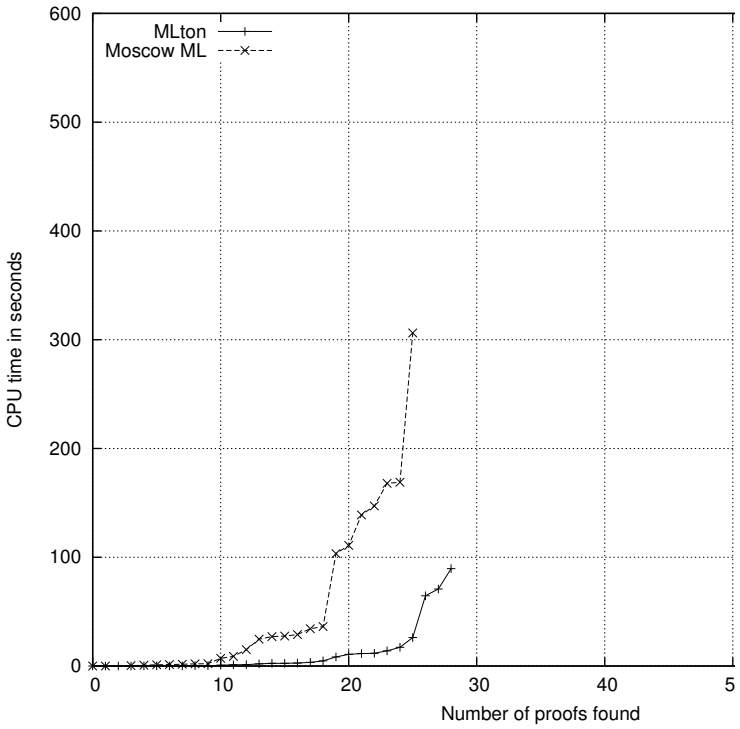


Fig. 1. The times at which the provers discovered proofs.

References

1. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gérard Huet, César A. Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, INRIA (Institut National de Recherche en Informatique et en Automatique), France, 1997.
2. Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, and Keith Wansbrough. The TCP specification: A quick introduction. Available from Peter Sewell's web site, March 2004.
3. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.
4. Mike Gordon, Joe Hurd, and Konrad Slind. Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 200–215. Springer, October 2003.
5. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.
6. F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, July 1999. Springer.

Higher-Level Hardware Synthesis in HOL

Juliano Iyoda and Michael J. C. Gordon

University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, U.K.
{*Juliano.Iyoda,Mike.Gordon*}*@cl.cam.ac.uk*

Abstract. A simple functional language embedded in higher order logic is used as a hardware description language. Our approach uses proof scripts to synthesise circuits directly from logical specifications. As well as synthesising implementations, we also generate theorems exhibiting their correctness. Our goal is to experiment with synthesis by proof along a spectrum of automation ranging from push-button compilation to user guided refinement. This paper describes formal compilation to synchronous implementations with a handshaking interface.

1 Introduction

We describe an approach to hardware synthesis by mechanised proof. A compiler, implemented as a proof rule, transforms specifications expressed in a simple functional language embedded in higher order logic into hardware devices that interact via a handshaking protocol. This approach allows the designer to focus solely on the high level behaviour of the system without having to reason about the correctness of the circuit at the gate level.

Our compilation method is partly inspired by SAFL [10], especially ideas in Richard Sharp's PhD [12]. Our long term goal is to develop correct-by-construction SAFL-like formal synthesis by proof. The current paper is only a very first proof-of-concept step.

Higher order logic (HOL) [6] has already been successfully applied to specify and verify hardware [4, 5, 9], and functional programming languages have been used as hardware description languages [2, 11, 12]. Formal synthesis by proof has previously been investigated by, among others, Johnson and Bose [8], Hanna [7], Fourman [3] and a researchers at Karlsruhe on high-level synthesis using the *Gropius* language [1, 13].

The novelty of our work is (i) the details of the device interface, (ii) the implementation of synthesis by deduction (rather than by the application of pre-verified transformations) and (iii) the way synthesis results are encoded as composable theorems certifying the correctness of the synthesised implementations.

Section 2 introduces the simple functional language used as source code. Section 3 defines the specification of generic handshaking devices to be used during the compilation.

The implementation and the verification of handshaking devices are presented in Section 4. The synthesis-by-proof algorithm is described in Section 5 and is illustrated by a case study in Section 6. Finally, conclusions and future work are outlined in Section 7.

2 A Simple Language

In collaboration with Konrad Slind of the University of Utah, we eventually plan to compile from an ML-like subset of higher order logic, but in this paper we start from an intermediate language consisting of expressions built using a set of simple operators. These are quite expressive, and the construction of a front end to parse into the intermediate language is orthogonal to the work described here.

We implement functions of type $\sigma_1 \times \cdots \times \sigma_m \rightarrow \tau_1 \times \cdots \times \tau_n$ where $\sigma_1, \ldots, \sigma_m, \tau_1, \ldots, \tau_n$ are types of values that can be carried on busses. In real applications, these types will often be words of different widths, but in this paper we will use booleans (T and F are the only values of type *bool*) and natural numbers ($0, 1, \ldots$ etc. of type *num*). Let $f, f_1, f_2, \ldots$ range over such functions. The constructs of our language are expressions e given in BNF by:

$$e ::= \mathsf{Atm}\ f \mid \mathsf{Lib}\ f \mid \mathsf{Seq}\ e_1\ e_2 \mid \mathsf{Par}\ e_1\ e_2 \mid \mathsf{Ite}\ e_1\ e_2\ e_3 \mid \mathsf{Rec}\ e_1\ e_2\ e_3$$

Both $\mathsf{Atm}\ f$ and $\mathsf{Lib}\ f$ implement function f. The difference is that $\mathsf{Atm}\ f$ is constructed from a combinational circuit (see definition of ATM in Section 4 below) and $\mathsf{Lib}\ f$ assumes f is in a library (initially assumed empty) of previously designed components (see Section 6.2 for an example). We make a shallow embedding of expressions in higher order logic by defining functions with the same names as the expression constructors by:

$$\begin{array}{ll}
\mathsf{Lib}\ f & = f \\
\mathsf{Atm}\ f & = f \\
\mathsf{Seq}\ f_1\ f_2 & = \lambda x.\ f_2(f_1\ x) \\
\mathsf{Par}\ f_1\ f_2 & = \lambda x.\ (f_1\ x,\ f_2\ x) \\
\mathsf{Ite}\ f_1\ f_2\ f_3 & = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f_3\ x \\
\mathsf{Rec}\ f_1\ f_2\ f_3 & = \epsilon f.\ f\ =\ \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f(f_3\ x)
\end{array}$$

$\mathsf{Rec}\ f_1\ f_2\ f_3$ uses Hilbert's ϵ-operator, and so means "choose a function f such that f satisfies the equation $f = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f(f_3\ x)$". In practise, f_1, f_2 and f_3 will be such that f is uniquely determined. For example, taking:

$$\begin{array}{l}
f_1 = \lambda(n, acc).\ n = 0 \\
f_2 = \lambda(n, acc).\ (n, acc) \\
f_3 = \lambda(n, acc).\ (n{-}1,\ n{\times}acc)
\end{array}$$

uniquely defines the function

$$f = \lambda(n, acc).\ \texttt{if}\ (n = 0)\ \texttt{then}\ (n, acc)\ \texttt{else}\ f(n{-}1,\ n{\times}acc)$$

A program p is a list of declarations $\langle c_1 = e_1\ \ldots\ c_n = e_n \rangle$, where for $1 \leq i \leq n$, c_i is a new name and e_i is an expression built out of library functions and $c_1, \ldots, c_{i-1}$.

3 Handshaking Devices

Our compiler takes a pair $(\langle c_1 = e_1 \ \ldots \ c_n = e_n \rangle, \ e)$, consisting of a program $\langle c_1 = e_1 \ \ldots \ c_n = e_n \rangle$ and expression e. It generates a clocked device that computes e via a simple handshaking protocol. This section describes the protocol and its definition in HOL.

Figure 1 shows a sequence of events that illustrates a transaction in which a handshaking device performs a single computation starting at a time t and ending at a later time t' (where time counts cycles). The variables *inp* and *out* represent the usual input and output data, respectively. The wires *load* and *done* control the access to the device. If *done* is asserted, it means that the device is idle and ready to compute a request. Once a positive edge on *load* is detected, the device samples the input and starts to compute the result (see when $(time = t)$ and $(time = t + 1)$ at Figure 1). During the computation, *done* remains low and every call is ignored. Eventually, the device outputs the result and indicates its completion by asserting *done*.

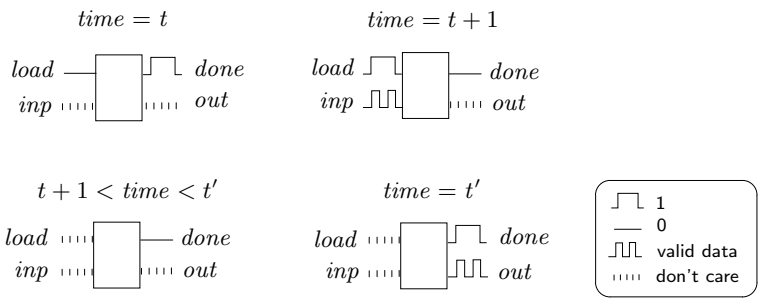


Fig. 1. A handshake protocol.

Suppose the device computes a function f. At the start of a transaction (time t) the device must be outputting T on *done* (to indicate it is ready) and the environment must be asserting F on *load* (i.e. in a state such that a positive edge on *load* can be generated). A transaction is initiated by asserting (at time $t{+}1$) the value T on *load* (i.e. *load* has a positive edge at time $t{+}1$), and this causes the device to read the value, v say, being input on *inp* (at time $t{+}1$) and to de-assert *done*. The device then becomes insensitive to inputs until T is next asserted on *done*, at which time (say time $t' > t{+}1$) the value $f(v)$ computed will be output on *out*.

The behaviour of hardware is modelled in HOL as a boolean-valued term whose free variables represent the external (observable) wires of the circuit. This term evaluates to true if the values observed at the external wires could occur in the circuit. The variables are functions from natural numbers (representing time) to values. For a signal, the low value zero and the high value one are represented by false (F) and true (T), respectively.

Before specifying the behaviour of a handshaking device, the auxiliary predicates Posedge and HoldF are defined.

A positive edge of a signal is defined as the transition of its value from low to high or, in our case, from F to T. Posedge is specified by:

$$\vdash \mathsf{Posedge}\ s\ t\ =\ \texttt{if}\ t{=}0\ \texttt{then}\ \mathsf{F}\ \texttt{else}\ (\neg\, s(t{-}1)\ \wedge\ s\ t\ )$$

Note that if the time is zero, it is assumed that no positive edge has occurred.

The term $\mathsf{HoldF}\ (t_1, t_2)\ s$ says that a signal s holds a low value F during a half-open interval starting at t_1 to just before t_2.

$$\vdash \mathsf{HoldF}\ (t_1, t_2)\ s = \forall t.\ t_1 \leq t < t_2\ \Rightarrow\ \neg(s\ t)$$

The behaviour of the handshaking device computing a function f is described by the term $\mathsf{Dev}\ f\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out})$ where:

$$\begin{array}{l}\vdash\ \mathsf{Dev}\ f\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) = \\ \quad (\forall t.\ \mathit{done}\ t\ \wedge\ \mathsf{Posedge}\ \mathit{load}\ (t{+}1) \\ \qquad \Rightarrow \\ \qquad \exists t'.\ t' > t{+}1\ \wedge\ \mathsf{HoldF}\ (t{+}1, t')\ \mathit{done}\ \wedge \\ \qquad\quad \mathit{done}\ t'\ \wedge\ (\mathit{out}\ t' = f(\mathit{inp}\ (t{+}1)))) \\ \quad \wedge \\ \quad (\forall t.\ \mathit{done}\ t\ \wedge\ \neg(\mathsf{Posedge}\ \mathit{load}\ (t{+}1))\ \Rightarrow \mathit{done}\ (t{+}1))\end{array}$$

The first conjunct in the right-hand side describes the context presented in Figure 1. If the device is available and a positive edge occurs on *load*, there exists a time t' in future when *done* signals its termination and the output is produced. The value of the output at time t' is the result of applying f to the value of the input at time $t{+}1$. The signal *done* holds the value F during the computation. The second conjunct specifies the situation where no call is made on *load* and the device simply remains idle.

4 Implementing Handshaking Devices

This section describes how we implement our language. Our convention is to use fully capitalised named for primitive circuits and circuit constructors. First, we describe a circuit constructor ATM that builds a handshaking device from a combinational circuit. Next we describe circuit constructors SEQ, PAR, ITE and REC that compose handshaking devices corresponding to Seq $e_1\ e_2$, Par $e_1\ e_2$, Ite $e_1\ e_2\ e_3$ and Rec $e_1\ e_2\ e_3$, respectively. The key property of these constructors that ensures they are correct are the following theorems (the notation $g \circ f$ denotes the function composition $\lambda x.\ g(f\ x)$):

$$\begin{array}{l}\vdash\ \mathsf{ATM}\ f\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) \\ \quad \Rightarrow\ \mathsf{Dev}\ f\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out})\end{array}$$

$$\begin{array}{l}\vdash\ \mathsf{SEQ}\ (\mathsf{Dev}\ f_1)\ (\mathsf{Dev}\ f_2)\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) \\ \quad \Rightarrow\ \mathsf{Dev}\ (f_2 \circ f_1)\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out})\end{array}$$

$$\vdash \ \mathsf{PAR}\ (\mathsf{Dev}\ f_1)\ (\mathsf{Dev}\ f_2)\ (load, inp, done, out)$$
$$\Rightarrow\ \mathsf{Dev}\ (\lambda\, x.\ (f_1\ x, f_2\ x))\ (load, inp, done, out)$$

$$\vdash \ \mathsf{ITE}\ (\mathsf{Dev}\ f_1)\ (\mathsf{Dev}\ f_2)\ (\mathsf{Dev}\ f_3)\ (load, inp, done, out)$$
$$\Rightarrow\ \mathsf{Dev}\ (\lambda\, x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f_3\ x)\ (load, inp, done, out)$$

$$\vdash \ \mathsf{Total}(f_1, f_2, f_3) \wedge \mathsf{REC}\ (\mathsf{Dev}\ f_1)\ (\mathsf{Dev}\ f_2)\ (\mathsf{Dev}\ f_3)\ (load, inp, done, out)$$
$$\Rightarrow\ \mathsf{Dev}\ (\mathsf{Rec}\ f_1\ f_2\ f_3)\ (load, inp, done, out)$$

where $\mathsf{Total}(f_1, f_2, f_3)$ is a predicate ensuring that there is a unique function satisfying $f = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f(f_3\ x)$ and is defined by:

$$\mathsf{Total}(f_1, f_2, f_3)\ =\ \exists variant.\ \forall x.\ \neg(f_1\ x) \Rightarrow variant(f_3\ x) < variant\ x$$

The constructors ATM, SEQ, PAR, ITE and REC use some primitive combinational hardware components AND, OR, NOT and MUX, and two primitive sequential components DEL and DFF. The behaviour of a combinational AND-gate is specified as a relation that constrains the value of the output to the conjunction of the inputs.

$$\vdash\ \mathsf{AND}\ (in_1, in_2, out) = \forall t.\ out\ t = (in_1\ t\ \wedge\ in_2\ t)$$

A combinational OR-gate with inputs in_1 and in_2 and output out is defined in a similar way.

$$\vdash\ \mathsf{OR}\ (in_1, in_2, out) = \forall t.\ out\ t = (in_1\ t\ \vee\ in_2\ t)$$

An inverter simply outputs the negation of the input.

$$\vdash\ \mathsf{NOT}\ (inp, out) = \forall t.\ out\ t = \neg(inp\ t)$$

A multiplexer connects the input in_1 to the output out if the selector sel has the value T. Otherwise, it outputs the value of in_2.

$$\vdash\ \mathsf{MUX}\ (sel, in_1, in_2, out) = \forall t.\ out\ t = \texttt{if}\ sel\ t\ \texttt{then}\ in_1\ t\ \texttt{else}\ in_2\ t$$

In general, a combinational component computing a function f is specified by:

$$\vdash\ \mathsf{COMB}\ f\ (inp, out) = \forall t.\ out\ t = f(inp\ t)$$

At any given time, this generic combinational device outputs f applied to the current value of the input.

A delay outputs the value of the input at the previous time.

$$\vdash\ \mathsf{DEL}\ (inp, out) = (out\ 0 = inp\ 0)\ \wedge\ (\forall t.\ out(t{+}1) = inp\ t)$$

At time zero, the delay behaves as a wire. A D-type flip-flop DFF outputs the value of the input d on the positive edge of the signal clk. If no positive edge occurs, the output q remains unchanged.

$$\vdash\ \mathsf{DFF}(d, clk, q)\ =\ \forall t.\ q(t{+}1) = \texttt{if}\ \mathsf{Posedge}\ clk\ (t{+}1)\ \texttt{then}\ d(t{+}1)\ \texttt{else}\ q\ t$$

The connection between two components is modelled by the conjunction of their specifications. The physical connection is represented by the identically-labelled wires of the subcomponents. Moreover, the existential quantifier hides the internal wires of the composite device.

For example, $\mathsf{POSEDGE}(inp, out)$ specifies a composite device that asserts T on its output *out* if and only if a positive edge has occurred on the input *inp*. Our implementation is:

$$\vdash\ \mathsf{POSEDGE}(inp, out) = \exists c_0\ c_1.\ \mathsf{DEL}(inp, c_0) \wedge \mathsf{NOT}(c_0, c_1) \wedge \mathsf{AND}(c_1, inp, out)$$

This component connects a DEL, NOT and AND by the internal wires c_0 and c_1. The wire c_0 has the value of the input at the previous time. The circuit outputs T if c_0 has the value F and the current input is T — which is exactly what characterises a positive edge. It is easy to show that POSEDGE has the following property.

$$\vdash\ \mathsf{POSEDGE}(inp, out)\ \Rightarrow\ (\forall t.\ out\ t = \mathsf{Posedge}\ inp\ t)$$

The circuit ATM implements an atomic device.

$$\begin{array}{l}\vdash\ \mathsf{ATM}\ f\ (load, inp, done, out) = \\ \quad \exists c_0\ c_1.\ \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{NOT}(c_0, done)\ \wedge \\ \qquad \mathsf{COMB}\ f\ (inp, c_1)\ \wedge\ \mathsf{DEL}(c_1, out)\end{array}$$

This device takes one time unit to compute (see Figure 2(a)). Although a combinational circuit is clearly more efficient than an atomic device, this device is suitable for composing with other handshaking devices.

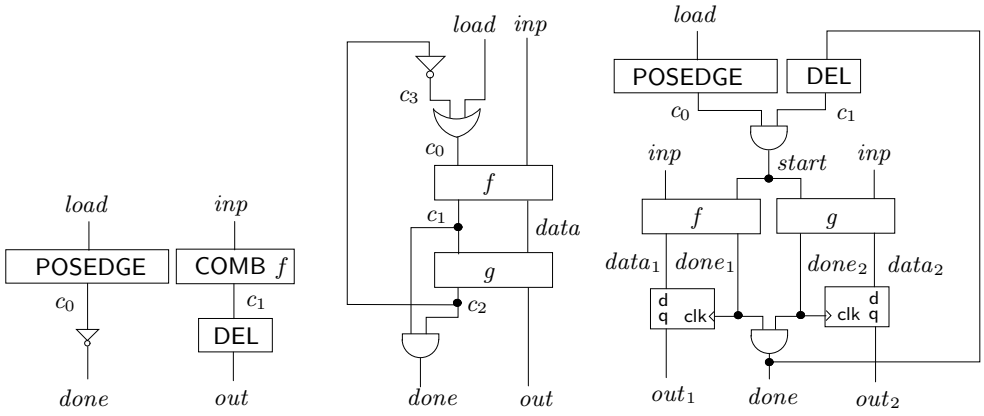
The constructor SEQ specifies a circuit which combines two devices to compute in sequence.

$$\begin{array}{l}\vdash\ \mathsf{SEQ}\ f\ g\ (load, inp, done, out) = \\ \quad \exists c_0\ c_1\ c_2\ c_3\ data. \\ \qquad \mathsf{NOT}(c_2, c_3)\ \wedge\ \mathsf{OR}(c_3, load, c_0)\ \wedge\ f(c_0, inp, c_1, data)\ \wedge \\ \qquad g(c_1, data, c_2, out)\ \wedge\ \mathsf{AND}(c_1, c_2, done)\end{array}$$

The subcomponents f and g have the same interface of a handshaking device. The output of the component f is the input of the component g (see the variables c_1 and *data* in Figure 2(b)). This composite device signals its completion when both f and g terminate.

The constructor PAR combines two devices in parallel.

$$\begin{array}{l}\vdash\ \mathsf{PAR}\ f\ g\ (load, inp, done, out) = \\ \quad \exists c_0\ c_1\ start\ done_1\ done_2\ data_1\ data_2\ out_1\ out_2. \\ \qquad \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{DEL}(done, c_1)\ \wedge\ \mathsf{AND}(c_0, c_1, start)\ \wedge \\ \qquad f(start, inp, done_1, data_1)\ \wedge\ g(start, inp, done_2, data_2)\ \wedge \\ \qquad \mathsf{DFF}(data_1, done_1, out_1)\ \wedge\ \mathsf{DFF}(data_2, done_2, out_2)\ \wedge \\ \qquad \mathsf{AND}(done_1, done_2, done)\ \wedge\ (out = \lambda t.\ (out_1\ t, out_2\ t))\end{array}$$



(a) ATM f (b) SEQ f g (c) PAR f g

Fig. 2. Implementation of composite devices.

The devices f and g are triggered simultaneously by *start* and return $data_1$ and $data_2$, respectively (see Figure 2(c)). As f and g may terminate at different times, their outputs are stored by DFFs and made available by out_1 and out_2. The components POSEDGE and DEL prevent calls to either f or g during their computation.

The conditional constructor ITE implements an if-then-else circuit from three subcomponents.

$$
\begin{aligned}
\vdash\ & \mathsf{ITE}\ e\ f\ g\ (load, inp, done, out) = \\
& \exists c_0\ c_1\ c_2\ start\ start'\ done\_e\ data\_e\ q\ not\_e\ data\_f\ data\_g\ sel \\
& \quad done\_f\ done\_g\ start\_f\ start\_g. \\
& \quad \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{DEL}(done, c_1)\ \wedge\ \mathsf{AND}(c_0, c_1, start)\ \wedge \\
& \quad e(start, inp, done\_e, data\_e)\ \wedge\ \mathsf{POSEDGE}(done\_e, start')\ \wedge \\
& \quad \mathsf{DFF}(data\_e, done\_e, sel)\ \wedge\ \mathsf{DFF}(inp, start, q)\ \wedge \\
& \quad \mathsf{AND}(start', data\_e, start\_f)\ \wedge\ \mathsf{NOT}(data\_e, not\_e)\ \wedge \\
& \quad \mathsf{AND}(start', not\_e, start\_g)\ \wedge\ f(start\_f, q, done\_f, data\_f)\ \wedge \\
& \quad g(start\_g, q, done\_g, data\_g)\ \wedge\ \mathsf{MUX}(sel, data\_f, data\_g, out)\ \wedge \\
& \quad \mathsf{AND}(done\_e, done\_f, c_2)\ \wedge\ \mathsf{AND}(c_2, done\_g, done)
\end{aligned}
$$

The device e implements a boolean test, while f and g implement the conditional branches. The output of e triggers either f or g (see the variable $data\_e$ in Figure 3). A multiplexer selects the right output based on the (stored) value of $data\_e$. The variable *done* is asserted if all subcomponents have terminated.

A function is tail-recursive if its recursive calls are the very last executed statements in the function. Tail-recursion is interesting for hardware compilation because it does not require the compiler to allocate storage for every function call.

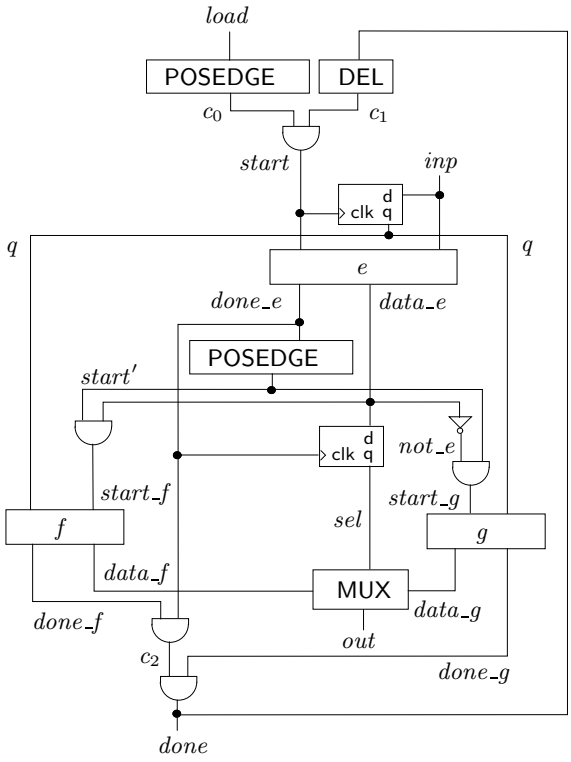


Fig. 3. The conditional constructor: ITE e f g.

The language introduced in Section 2 has an operator Rec for specifying tail-recursive functions $\mathcal{F}$ of the form

$$\mathcal{F}\ x\ =\ \texttt{if}\ e\ x\ \texttt{then}\ f\ x\ \texttt{else}\ \mathcal{F}(g\ x)$$

Such a function $\mathcal{F}$ is specified by Rec e f g as defined above. A handshaking circuit that implements $\mathcal{F}$ (if it is well-defined) is constructed using the REC constructor, where:

$$\begin{array}{l}\vdash\ \mathsf{REC}\ e\ f\ g\ (load, inp, done, out) = \\ \quad \exists done\_g\ data\_g\ start\_e\ q\ done\_e\ data\_e\ start\_f\ start\_g\ inp\_e\ done\_f \\ \quad\ c_0\ c_1\ c_2\ c_3\ c_4\ start\ sel\ start'\ not\_e. \\ \qquad \mathsf{POSEDGE}(load, c_0)\ \wedge\ \mathsf{DEL}(done, c_1)\ \wedge\ \mathsf{AND}(c_0, c_1, start)\ \wedge \\ \qquad \mathsf{OR}(start, sel, start\_e)\ \wedge\ \mathsf{POSEDGE}(done\_g, sel)\ \wedge \\ \qquad \mathsf{MUX}(sel, data\_g, inp, inp\_e)\ \wedge\ \mathsf{DFF}(inp\_e, start\_e, q)\ \wedge \\ \qquad e(start\_e, inp\_e, done\_e, data\_e)\ \wedge\ \mathsf{POSEDGE}(done\_e, start')\ \wedge \\ \qquad \mathsf{AND}(start', data\_e, start\_f)\ \wedge\ \mathsf{NOT}(data\_e, not\_e)\ \wedge \\ \qquad \mathsf{AND}(not\_e, start', start\_g)\ \wedge\ f(start\_f, q, done\_f, out)\ \wedge \\ \qquad g(start\_g, q, done\_g, data\_g)\ \wedge\ \mathsf{DEL}(done\_g, c_3)\ \wedge \\ \qquad \mathsf{AND}(done\_g, c_3, c_4)\ \wedge\ \mathsf{AND}(done\_f, done\_e, c_2)\ \wedge\ \mathsf{AND}(c_2, c_4, done)\end{array}$$

The recursive constructor is similar to the conditional one (see Figure 4). The main difference is the connection between the "else" branch and the circuit itself — characterising a recursive call. A multiplexer selects the input from either the external environment or from the recursive call. The circuit terminates if every subcomponent terminates (see the variables *done_e*, *done_f* and *done_g* in Figure 4). Furthermore, the component g must have terminated at least one time unit before. This is necessary to distinguish a recursive call from the complete termination of the computation.

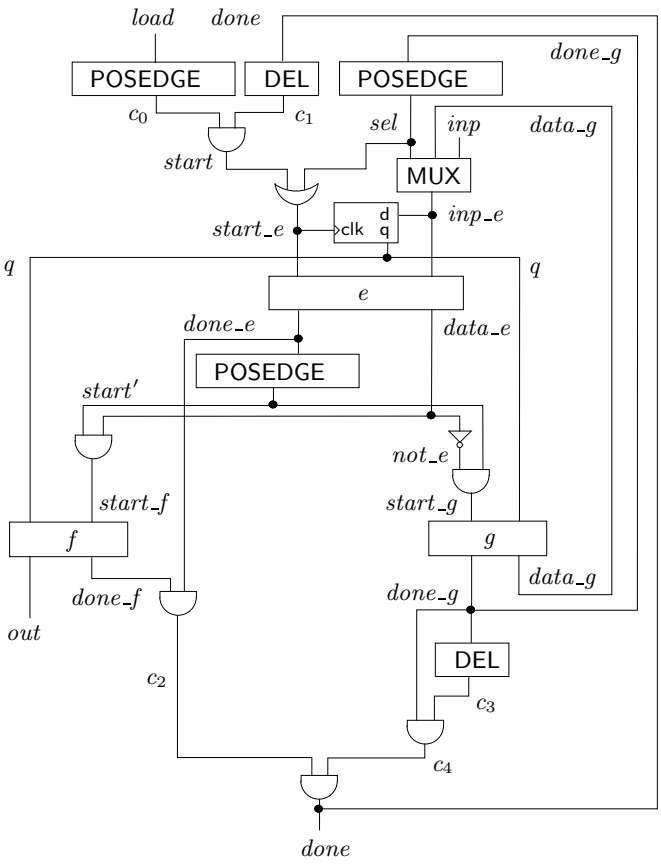


Fig. 4. The recursive constructor: REC e f g.

5 Compiling by Proof

The prototype compiler we have implemented takes a program and an expression $(\langle c_1 = e_1 \ \ldots \ c_n = e_n \rangle,\ e)$, where the expression e is built out of library functions

and $c_1, \ldots, c_n$. It generates a circuit $\mathcal{C}(load, inp, done, out)$, represented as a term in higher order logic, and returns a theorem:

$$\vdash\ \forall load\ inp\ done\ out.\ \mathcal{C}(load, inp, done, out) \Rightarrow \mathsf{Dev}\ e\ (load, inp, done, out)$$

The compilation procedure is a straightforward recursive application of the following theorems (which are proved from the key properties given in Section 4 and the semantics of the expression and circuit constructors):

`ATM_INTRO`

$$\vdash\ \forall c\ s.\ \mathsf{ATM}\ c\ s\ \Rightarrow\ \mathsf{Dev}\ c\ s$$

`SEQ_INTRO`

$$\begin{array}{l}\vdash\ \forall P_1\ P_2\ f_1\ f_2.\\ \quad (\forall s.\ P_1\ s\ \Rightarrow\ \mathsf{Dev}\ f_1\ s)\ \wedge\ (\forall s.\ P_2\ s\ \Rightarrow\ \mathsf{Dev}\ f_2\ s)\\ \quad \Rightarrow\\ \quad \forall s.\ \mathsf{SEQ}\ P_1\ P_2\ s\ \Rightarrow\ \mathsf{Dev}\ (\mathsf{Seq}\ f_1\ f_2)\ s\end{array}$$

`PAR_INTRO`

$$\begin{array}{l}\vdash\ \forall P_1\ P_2\ f_1\ f_2.\\ \quad (\forall s.\ P_1\ s\ \Rightarrow\ \mathsf{Dev}\ f_1\ s)\ \wedge\ (\forall s.\ P_2\ s\ \Rightarrow\ \mathsf{Dev}\ f_2\ s)\\ \quad \Rightarrow\\ \quad \forall s.\ \mathsf{PAR}\ P_1\ P_2\ s\ \Rightarrow\ \mathsf{Dev}\ (\mathsf{Par}\ f_1\ f_2)\ s\end{array}$$

`ITE_INTRO`

$$\begin{array}{l}\vdash\ \forall P_1\ P_2\ P_3\ f_1\ f_2\ f_3.\\ \quad (\forall s.\ P_1\ s\ \Rightarrow\ \mathsf{Dev}\ f_1\ s)\ \wedge\\ \quad (\forall s.\ P_2\ s\ \Rightarrow\ \mathsf{Dev}\ f_2\ s)\ \wedge\\ \quad (\forall s.\ P_3\ s\ \Rightarrow\ \mathsf{Dev}\ f_3\ s)\\ \quad \Rightarrow\\ \quad \forall s.\ \mathsf{ITE}\ P_1\ P_2\ P_3\ s\ \Rightarrow\ \mathsf{Dev}\ (\mathsf{Ite}\ f_1\ f_2\ f_3)\ s\end{array}$$

`REC_INTRO`

$$\begin{array}{l}\vdash\ \forall f_1\ f_2\ f_3\ P_1\ P_2\ P_3.\\ \quad \mathsf{Total}(f_1, f_2, f_3)\\ \quad \Rightarrow\\ \quad (\forall s.\ P_1\ s\ \Rightarrow\ \mathsf{Dev}\ f_1\ s)\ \wedge\\ \quad (\forall s.\ P_2\ s\ \Rightarrow\ \mathsf{Dev}\ f_2\ s)\ \wedge\\ \quad (\forall s.\ P_3\ s\ \Rightarrow\ \mathsf{Dev}\ f_3\ s)\\ \quad \Rightarrow\\ \quad \forall s.\ \mathsf{REC}\ P_1\ P_2\ P_3\ s\ \Rightarrow\ \mathsf{Dev}\ (\mathsf{Rec}\ f_1\ f_2\ f_3)\ s\end{array}$$

The theorem `REC_INTRO` is an implication whose antecedent is $\mathsf{Total}(f_1, f_2, f_3)$.

We will outline how our compiler works using an ML-style pseudo-code to describe the inferences that deductively transform a specification to an implementation. Theorems in the HOL system logic have the form $\Gamma \vdash t$ where Γ is a set of assumptions and t is a conclusions that follows from the assumptions.

The ML pseudo-code `SPEC` $[t_1, \ldots, t_n]$ $(\Gamma \vdash \forall x_1 \ldots x_n.\ P(x_1, \ldots, x_n))$ evaluates to $\Gamma \vdash P(t_1, \ldots, t_n)$. `UNDISCH`$(\Gamma \vdash t_1 \Rightarrow t_2)$ evaluates to $\Gamma \cup \{t_1\} \vdash t_2$. `MATCH_MP` $(\Gamma_1 \vdash t)$ $(\Gamma_2 \vdash t_1 \Rightarrow t_2)$ matches t_1 with t and then instantiates the theorem $(\Gamma_2 \vdash t_1 \Rightarrow t_2)$ to $(\Gamma_2 \vdash t \Rightarrow t')$ (where t' is the instance of t_2 corresponding to the match) and returns $\Gamma_1 \cup \Gamma_2 \vdash t'$, the result of applying Modus Ponens. $(\Gamma_1 \vdash t_1)$ `AND` $(\Gamma_2 \vdash t_2)$ evaluates to $(\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2)$. Evaluating `LibraryLookup lib` f searches the library `lib` for a theorem of the form:

$$\vdash\ \forall load\ inp\ done\ out.\ \mathcal{C}(load, inp, done, out) \Rightarrow \mathsf{Dev}\ f\ (load, inp, done, out)$$

and uses the first one it finds (or raises an exception if no matching theorem found in `lib`).

To compile $(p,$ `e`$)$, first rewrite `e` with the definitions in p to obtain an expanded expression `e'` that only contains atomic or library functions, and then recursively apply the rules below to evaluate `Compile e'`.

```
Compile lib (Atm f) =
 SPEC f ATM_INTRO

Compile lib (Lib f) =
 LibraryLookup lib f

Compile lib (Seq e1 e2) =
 MATCH_MP SEQ_INTRO (Compile lib e1 AND Compile lib e2)

Compile lib (Par e1 e2) =
 MATCH_MP PAR_INTRO (Compile lib e1 AND Compile lib e2)

Compile lib (Ite e1 e2 e3) =
 MATCH_MP
  PAR_INTRO
  (Compile lib e1 AND Compile lib e2 AND Compile lib e3)

Compile lib (Rec e1 e2 e3) =
 MATCH_MP
  (UNDISCH(SPEC [e1,e2,e3] REC_INTRO))
  (Compile lib e1 AND Compile lib e2 AND Compile lib e3)
```

Note that evaluating `Compile lib (Rec e1 e2 e3)` will generate a theorem with an assumption $\mathsf{Total}(\mathtt{e1}, \mathtt{e2}, \mathtt{e3})$.

6 The Factorial Case Study

The tail-recursive function FactIter defined below can be used to compute the factorial function.

$$\vdash\ \mathsf{FactIter}(n, acc) = \mathtt{if}\ (n = 0)\ \mathtt{then}\ (n, acc)\ \mathtt{else}\ \mathsf{FactIter}(n{-}1, n{\times}acc)$$

The variable acc accumulates the result of the computation. Evaluating $\mathsf{FactIter}(n, 1)$ returns $(0,\ n!)$, where $n!$ is the factorial of n.

6.1 Implementation with an atomic (combinational) multiplier

The following program in our language computes $n!$.

$$
\begin{array}{rcl}
\mathit{FactProg} & = & \langle \mathsf{Test0} \quad = \mathsf{Atm}\ \lambda n.\ n = 0, \\
& & \ \ \mathsf{Ident} \quad = \mathsf{Atm}\ \lambda(n, acc).\ (n, acc), \\
& & \ \ \mathsf{Dec} \quad = \mathsf{Atm}\ \lambda n.\ n{-}1, \\
& & \ \ \mathsf{Mult} \quad = \mathsf{Atm}\ \lambda(n, acc).\ n{\times}acc, \\
& & \ \ \mathsf{Fst} \quad = \mathsf{Atm}\ \lambda(n, acc).\ n, \\
& & \ \ \mathsf{Snd} \quad = \mathsf{Atm}\ \lambda(n, acc).\ acc, \\
& & \ \ \mathsf{PairOne} = \mathsf{Atm}\ \lambda n.\ (n, 1), \\
& & \ \ \mathsf{FactIter} = \mathsf{Rec}\ (\mathsf{Seq}\ \mathsf{Fst}\ \mathsf{Test0})\ \mathsf{Ident}\ (\mathsf{Par}\ (\mathsf{Seq}\ \mathsf{Fst}\ \mathsf{Dec})\ \mathsf{Mult}), \\
& & \ \ \mathsf{Fact} \quad = \mathsf{Seq}\ \mathsf{PairOne}\ (\mathsf{Seq}\ \mathsf{FactIter}\ \mathsf{Snd})\rangle
\end{array}
$$

The expressions Test0, Ident, Dec, Mult, Fst, Snd and PairOne are assumed atomic (i.e. implementable by combinational circuits). This is unrealistic for Mult; see Section 6.2 for a (slightly) more realistic version.

If we invoke the compiler on the program ($\mathit{FactProg}$, Fact) the result is:

```
[ TOTAL (Seq (λ(n,acc). n) (λn. n = 0), (λ(n,acc). (n,acc)),
         Par (Seq (λ(n,acc). n) (λn. n-1)) (λ(n,acc). n×acc)) ]
    |- ∀load inp done out.
         SEQ (ATM (λn. (n,1)))
           (SEQ
              (REC (SEQ (ATM (λ(n,acc). n)) (ATM (λn. n = 0)))
                 (ATM (λ(n,acc). (n,acc)))
                 (PAR
                    (SEQ (ATM (λ(n,acc). n))
                       (ATM (λn. n-1)))
                    (ATM (λ(n,acc). n×acc))))
              (ATM (λ(n,acc). acc))) (load,inp,done,out)
         ⇒
         Dev Fact (load,inp,done,out)
```

The outcome is a theorem of the form $\Gamma \vdash t$ where Γ is a singleton set consisting of an assumption expressing the totality of FactIter. Simplifying the assumption with the definitions of Seq and Par yields:

$$\mathsf{Total}((\lambda(n, acc).\ n = 0), (\lambda(n, acc).\ (n, acc)), (\lambda(n, acc).\ (n{-}1, n{\times}acc)))$$

which is easily proved (with the function $(\lambda(x, y).\ x)$ as the variant). Once the totality assumption has been proved it can be eliminated. Furthermore, it is easy to prove by elementary arithmetic from the definitions of the components of $\mathit{FactProg}$ and the meanings of Atm, Seq, Par, Ite and Rec, that $\vdash \mathit{Fact} = \lambda n.\ n!$. The output of the compiler thus simplifies to:

```
|- ∀load inp done out.
        SEQ (ATM (λn. (n,1)))
          (SEQ
             (REC (SEQ (ATM (λ(n,acc). n)) (ATM (λn. n = 0)))
                (ATM (λ(n,acc). (n,acc)))
                (PAR
                   (SEQ (ATM (λ(n,acc). n))
                      (ATM (λn. n-1)))
                   (ATM (λ(n,acc). n×acc))))
             (ATM (λ(n,acc). acc))) (load,inp,done,out)
        ⇒
        Dev (λn. n!) (load,inp,done,out)
```

6.2 Implementation with a pre-verified multiplier

The example above used Mult = Atm $\lambda(n, acc).\ n{\times}acc$. Such a combinational multiplier is unrealistic (except for small words). However, we can easily implement a (naive) sequential multiplier that works by repeated addition and so, more realistically, only assumes combinational addition (and decrementing):

MultProg =
⟨Test0 = Atm $\lambda m.\ m = 0$,
Ident = Atm $\lambda(m, n, acc).\ (m, n, acc)$,
Dec = Atm $\lambda m.\ m{-}1$,
AddAcc = Atm $\lambda(m, n, acc).\ n{+}acc$,
Fst = Atm $\lambda(m, n, acc).\ m$,
Snd = Atm $\lambda(m, n, acc).\ n$,
Thd = Atm $\lambda(m, n, acc).\ acc$,
PairZero = Atm $\lambda(m, n).\ (m, n, 0)$,
MultIter = Rec (Seq Fst Test0) Ident (Par (Seq Fst Dec) (Par Snd AddAcc)),
Mult = Seq PairZero (Seq MultIter Thd)⟩

Note that we have used the same names in FactProg and MultProg for different (though semantically related) expressions (e.g. Fst). This is not a problem as names are local to the program they occur in.

Compiling (*MultProg*, Mult), simplifying and discharging the totality proof obligation (in a way very similar to the factorial example) results in:

```
|- ∀load inp done out.
     SEQ (ATM_IMP (λ(m,n). (m,n,0)))
       (SEQ
          (REC (SEQ (ATM_IMP (λ(m,n,acc). m)) (ATM_IMP (λm. m = 0)))
             (ATM_IMP (λ(m,n,acc). (m,n,acc)))
             (PAR (SEQ (ATM_IMP (λ(m,n,acc). m)) (ATM_IMP (λm. m - 1)))
                (PAR (ATM_IMP (λ(m,n,acc). n))
                   (ATM_IMP (λ(m,n,acc). n + acc)))))
          (ATM_IMP (λ(m,n,acc). acc))) (load,inp,done,out) ⇒
     Dev (λ(m,n). m × n) (load,inp,done,out)
```

After adding this theorem to the library, we can replace the combinational multiplier in the factorial example by Mult = Lib $\lambda(n, acc).\ n{\times}acc$.

If we recompile the factorial program after this change, the implementation of the multiplier is 'inlined' and we get:

```
|- ∀load inp done out.
     SEQ (ATM_IMP (λn. (n,1)))
       (SEQ
          (REC (SEQ (ATM_IMP (λ(n,acc). n)) (ATM_IMP (λn. n = 0)))
             (ATM_IMP (λ(n,acc). (n,acc)))
             (PAR (SEQ (ATM_IMP (λ(n,acc). n)) (ATM_IMP (λn. n - 1)))
                (SEQ (ATM_IMP (λ(m,n). (m,n,0)))
                   (SEQ
                      (REC
                         (SEQ (ATM_IMP (λ(m,n,acc). m))
                            (ATM_IMP (λm. m = 0)))
                         (ATM_IMP (λ(m,n,acc). (m,n,acc)))
                         (PAR
                            (SEQ (ATM_IMP (λ(m,n,acc). m))
                               (ATM_IMP (λm. m - 1)))
                            (PAR (ATM_IMP (λ(m,n,acc). n))
                               (ATM_IMP (λ(m,n,acc). n + acc)))))
                      (ATM_IMP (λ(m,n,acc). acc))))))
          (ATM_IMP (λ(n,acc). acc))) (load,inp,done,out) ⇒
     Dev (λn. n!) (load,inp,done,out)
```

This is an implementation of the factorial with an 'inner-loop' for each multiplication. Not an efficient circuit, but it illustrates hierarchical development.

7 Future Work

The handshaking protocol for devices is preliminary and we plan to refine and extend it. For example, Dev f $(load, inp, done, out)$ holds if F is continuously asserted on *done*. We need to prove some liveness results saying that if there is no posedge on *load* then eventually *done* will go to T. This property looks clearly true of ATM and should be compositional with respect to Seq, Par, Ite and Rec (assuming totality). The compiler should also be able to generate handshaking devices that are shared by several callers. An arbiter would control the concurrent calls and preserve the handshaking behaviour. This may require us to extend the handshaking protocol to support more than one request (*load*) and acknowledge (*done*) line per device.

In the future we plan to explore formally validated optimisations to the compiler, perhaps using ideas from SAFL compilation [12].

Finally, the compiler could provide the choice to generate either machine code or pure hardware. This feature would allow the user to partition the system into software and hardware parts and explore different designs.

8 Acknowledgements

Konrad Slind provided encouragement and motivation and also helped us with the use of TFL and the formulation and proof of REC_INTRO.

References

1. Christian Blumenroehr and Dirk Eisenbiegler. Performing High-Level Synthesis via Program Transformations within a Theorem Prover. In *Proceedings of the Digital System Design Workshop at the Euromicro 98 Conference, Västeras, Sweden*, pages 34–37, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1998. Online at: http://www.ubka.uni-karlsruhe.de/cgi-bin/psgunzip/1998/informatik/37/37.pdf.
2. Koen Claessen and Gordon Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits 2002 (DCC 2002)*, Grenoble, France, 2002.
3. S. Finn, M. Fourman, M. Francis, and R. Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110, Houthalen, Belgium, November 1989. Elsevier Science Publishers, B.V. North-Holland, Amsterdam.
4. Anthony C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, The Computer Laboratory, University of Cambridge, England, March 2001.
5. Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.
6. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. F. K. Hanna, N. Daeche, and W. G. J. Howells. Implementation of the Veritas Design Logic. In *Proc. Theorem Provers in Circuit Design*, pages 77–94. North Holland, 1992.
8. Steven D. Johnson and Bhaskar Bose. DDD – A System for Mechanized Digital Design Derivation. Technical Report TR323, Indiana University, IU Computer Science Department, 1990. Available on the Internet at: http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR323.
9. Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, Cambridge, England, 1993. Cambridge Tracts in Theoretical Computer Science 31.
10. Alan Mycroft and Richard Sharp. Hardware/software co-design using functional languages. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 236–251, Genova, Italy, April 2001. Springer-Verlag. LNCS Vol. 2031.
11. John O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382, Amsterdam, April 1987. North-Holland.
12. Richard Sharp. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, the Computer Laboratory, Cambridge, England, 2002.
13. C. Blumenröhr V. Sabelfeld and K. Kapp. Semantics and Transformations in Formal Synthesis at System Level. In Dines Bjorner, Manfred Broy, and Alexandre Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, Novosibirsk, Russia*, LNCS 2244, pages 149–156, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 2001. Springer-Verlag. http://link.springer.de/link/service/series/0558/bibs/2244/22440149.htm.

An Experiment in Automated Theorem Proving in Type Theory

Marcin Benke and Fredrik Lindblad

Department of Computing Science
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract We present some experiments in and tools for automated theorem proving in Martin-Löf's type theory. The main purpose of the tool is to facilitate interactive proving by filling out the tedious, yet relatively simple details of a proof. One particular application, where such automation is extremely handy, is verification of functional programs. The main contribution of this work is the use of induction and generalization in automated proofs, which is superior to automation available so far in proofs assistants based on higher-order logic, such as the Auto tactic in Coq.

1 Introduction

In this paper, we present some experiments in automated theorem proving in *Alfa* [HR00], an interactive proof system based on Martin-Löf type theory [CC99,ML84]. It can also be viewed as a small purely functional programming language with a type system that provides dependent types, thus allowing specification and verification of program properties within its type system. As such, Alfa supports algebraic datatypes, pattern matching and general recursive definitions. To preserve logical consistency, all proofs are subject to termination check as well as typechecking.

Alfa is a term-based proof editor. This means that the proof is presented and recorded as a term, rather than as a tactic expression as in tactic-based proof editors such as *Coq*.

Our tool, given a proposition (type), tries to find its proof (inhabitant). If a solution is found, it is presented as a proof term, which can be verified by the type-checker. The main purpose of the tool is to facilitate interactive proving by filling out the tedious, yet relatively simple details of a proof. One particular application, where such automation is extremely handy, is verification of functional programs.

The main contribution of this work is the use of induction and generalization in automated proofs, which is superior to automation available so far in proof assistants based on higher-order logic, such as the `Auto` tactic in *Coq*.

To ensure termination, we rely on structural induction. However, the tool can construct proofs with nested induction and induction appearing in subproofs as

well as generalize (strengthen) the induction hypothesis when needed. Proofs containing case analysis on compound expressions can sometimes be found. This is needed to prove e.g. that the functions *map* and *filter* commute in a certain way.

A notion of quasi normal forms has been investigated. One advantage of this approach is to avoid higher-order unification which is otherwise imposed in type theory. It can also reduce the amount of computation in the search. Although we can miss some equalities this way, tests have indicated that for most problems the fidelity of our notion of equality is sufficient. It also gives useful information as to when case analysis on compound expressions should be performed.

1.1 Related Work

A system for automatic theorem proving in Martin-Löf type theory has been developed by Tammet and Smith[TS98]. The implementation, called Gandalf, was designed to work with ALF, the predecessor of Agda. In Gandalf, problems are not solved directly. The authors mention the presence of higher-order unification as an obstacle of a direct approach. Instead, a theoretical basis for encoding ALF types into first-order intuitionistic logic is developed. Problems are encoded and then solved using various existing techniques for first-order logic. When a problem has been solved, the corresponding ALF term is constructed. Gandalf can produce inductive proofs. The encoding of such problems however seems to result in a rather time consuming search.

The Coq system for formalization and proof-checking is also based on an implementation of type theory. Proof search in Coq works with tactics. There are tactics for doing induction and generalization manually. There are also auto tactics which use elementary tactics in automatic proof search. These do however not make use of the tactics for induction and generalization, so such proofs cannot be found automatically.

2 The Language

The tool that has been developed is based on the Agda language[Coq98]. This subsection gives a brief description of the language and the notation. Only a fragment of Agda is used.

There are three type classes; function types, signatures and data types. The output type of a function may depend on the input value. The following notation is used for function types:

$$(x : X) \to Y$$

A signature consists of zero or more components with names. The type of a component may depend on the value of any previous component.

$$\mathbf{sig}\ \{(x_1 : X_1);\ \ldots; (x_n : X_n)\}$$

Data types consist of zero or more constructors, each taking any fixed number of arguments. An argument's type may depend on the values of previous arguments.

$$\begin{array}{l}\textbf{data}\ \mathsf{c}_1\ (x_{1,1} : X_{1,1})\ \ldots\ (x_{1,n_1} : X_{1,n_1}) \\ \quad\vdots \\ \mid \mathsf{c}_m\ (x_{m,1} : X_{m,1})\ \ldots\ (x_{m,n_m} : X_{m,n_m})\end{array}$$

The symbols '$\mathcal{S}$' and '$\mathcal{D}$' will designate the generic signature and data type as presented above. We let all type expressions themselves be of type $*$.

Functions are constructed with λ-abstractions:

$$\lambda(x : X) \to Y\ :\ (x : X) \to Y$$

The elements of signatures are structures:

$$\textbf{struct}\ \{x_1 = M_1;\ \ldots;\ x_n = M_n\}\ :\ \mathcal{S}$$

Elements of a data type are introduced by using any of its constructors.

$$\mathsf{c}_j\ M_1\ \ldots\ M_{n_j}\ :\ \mathcal{D}$$

The rule is valid for any $j \in [1, m]$.

Elimination is done by application, projection and case-expression respectively.

$$M\ N \qquad M.x_k \qquad \textbf{case}\ M\ \{\mathsf{c}_1\ \vec{y} \to N_1; \ldots; \mathsf{c}_m\ \vec{y} \to N_m\}$$

When type-checking a case-expression where the *scrutinee* is a parameter and not a compound expression, the parameter is substituted by its assumed value for each branch.

Reduction works in the expected way. Apart from these basic components, also the term construction **let** ... **in** ... will appear. It introduces local definitions and is used in the proof search for defining recursive functions. All occurrences will have the following form:

$$\begin{array}{l}\textbf{let}\ f\ (\vec{x} : \vec{X}) : T\ = M \\ \textbf{in}\ \ f\ \vec{N}\end{array}$$

Note that termination issues lie outside the typechecker. Termination check is performed as an extra step.

3 The Basic Concepts of the Proof Search

Proof search is performed mainly by backward reasoning. The basic concepts of the proof search are *target type*, *problem*, *meta variable* and *refinement*. The target type is the formulation of a proposition as a type. A problem is completely represented by a target type together with a variable environment in which an

element of the type is to be found. If such an element is found, it is a term proving the proposition. During the search, meta variables are used as place holders for subexpressions which are not yet known. These are denoted by question-marks. If we have an environment Γ and a target type T, then the initial problem of the search can be depicted like this:

$$\Gamma \vdash ? : T$$

The task is to find an expression not containing meta variables which can replace ?. This is done by successively replacing a meta variable by an expression containing new meta variables. The new meta variables correspond to the subproblems. The collection of partially given proof terms together with their associated subproblems are called the refinements of the problem.

The formulation of the initial problem must be extended to encompass all subproblems. The information of each new meta variable will be stored in a *meta environment*. For each meta variable, the meta environment gives the variable environment and the type associated to the meta variable. The letter Δ will be used for meta environments.

One more entity may appear in a problem. The way we handle data type elimination, solutions to subproblems are tagged with a set of *value constraints*. These are statements which specify which form a data type term must have in order to make the solution valid. In the end, no such constraints may of course remain for the solution of the main problem. Solutions with value constraints are combined in a case expression to generate a solution with less constraints. This is further explained in subsection 4.2. The value constraints have the form

$$M = \mathsf{c}_j \; x_{j,1} \; \ldots \; x_{j,n_j}$$

A set of value constraints will be denoted by σ.

We can now write the general form of a problem. First, there is a meta environment, then a variable context, Γ, which is associated to the meta variable of the subproblem. After this there may be a set of value constraints, which refer to variables in Γ. In this environment, a proof term should be found which has the same type as the current meta variable.

$$\Delta, \Gamma, \sigma \vdash ? \; : \; T$$

There are a number of different rules used by the tool to generate refinements, but they are all expressed in a uniform way to allow them to be processed equally. A refinement consists of a proof term. If the proof term is not complete, i.e. it contains new meta variables, the meta environment must be extended accordingly. New value constraints may also emerge and thus extend σ. In addition, the refinement can induce *meta variable constraints*. These have the form $? = M$. A meta variable constraint says that the refinement is valid only if the meta variable is bound to the specified term. Collections of meta variable constraints will be denoted by ρ.

The refinement rules will have this general form:

$$\frac{\Delta, \Gamma, \sigma \vdash ? \ : \ T \quad < \text{side conditions} >}{\Delta', \rho, \Gamma, \sigma' \vdash ? \rightsquigarrow M \ : \ T}$$

Refinements are recursively generated as long as new subproblems arise. When a refinement with no subproblems is encountered, it is a *solution* to the current problem. A refinement that does have subproblems is combined with the solutions of the subproblems, if such were found, to form a solution. Any solution that reach the top of the search potentially solves the given problem and contains all the necessary meta variable assignments to compile the proof term. Apart from the proof term information, solutions also inherit the meta variable and value constraints of the constituting refinements.

All new meta variables are classified as either proofs or parameters. Parameters are those which appear in other objects' types, i.e. those upon which other objects depend. The rest are considered to be proofs. Note that a parameter can depend on another parameter. The classification is used to determine which new meta variables should be treated as subproblems. Only proofs should be searched for. Searching for parameters is trivial and the result is arbitrary. The parameters' values should instead be settled as a side effect when searching for solutions to the proof objects.

Section 4 describes the collection of rules that are used to generate refinements.

3.1 Expression Reduction and Comparison

A notion of quasi normal forms for types has been used in the experiments. It is here called *simplification* of a type. When simplifying a type, computation is carried on until the next reduction step would introduce a non-reducible λ-abstraction, **case**-expression, **data**-expression or signature. This means that simplified types normally consist of applications where the terms are either identifiers, projections or λ-abstractions deriving from the type that was stated by the user. When a type is simplified, reduction is not only performed on the head of the expression. Also the subexpressions are reduced in the same way. This is necessary since there are refinement rules that do not just look at the head of the target type.

Using simplified types to do comparison entails the risk that two expressions that actually represent the same type, i.e. are convertible, are judged to be unequal. The simplification is designed to make most equal types syntactically equal modulo α-conversion. But in some situations this is not the case. To avoid this, a few technical things can be kept in mind when stating the problem. We also think that the simplification could be improved to achieve higher fidelity to actual equality.

One effect of type dependencies is that meta variables will occur in types. A meta variable represents an arbitrary value. This means that comparing two types involves assigning meta variables to values. Comparison is therefore replaced by unification. In type theory, higher-order unification is required. This

means that you have to take conversion equality into account when unifying. Higher-order unification is undecidable. Confer for instance [Dow01]. This problem is avoided here since syntactical equality is used and thus no reduction is allowed when unifying.

If the unification succeeds, the list of meta variable bindings is returned. The bindings are then used as constraints of the resulting refinement. Unification will be denoted as in the following example. Assume that two types, T_a and T_b, are unified in the meta environment Δ and variable environment Γ.

$$\Delta, (?_1 = M_1, ?_2 = M_2), \Gamma \vdash\ T_a \ \stackrel{\text{s}}{=}\ T_b$$

In the example, the unification induces the binding of two meta variables.

Types are however not only compared to each other. In some refinement rules ordinary head normalization is needed, denoted by the symbol '$\stackrel{n}{\twoheadrightarrow}$'. When normalization is used, the conclusion is always either of the following:

$$T \ \stackrel{\text{n}}{\twoheadrightarrow}\ (x : X) \to Y \qquad T \ \stackrel{\text{n}}{\twoheadrightarrow}\ \mathcal{S} \qquad T \ \stackrel{\text{n}}{\twoheadrightarrow}\ \mathcal{D}$$

4 Refinement Rules

This section presents most of the refinement rules which are implemented in the tool. There are rules for introduction, one for each type class, and one universal rule for elimination. Case analysis and induction are combined in one rule. Finally, one rule is devoted to generalization. Meta variables that are introduced in the rules are given various subscripts. It is however implicitly assumed that they all represent fresh meta variables.

4.1 Introduction Rules

The introduction rules construct a member of the target type according to the introduction typing rules of the language. There is one rule for each type class. First we have introduction of λ-abstractions:

$$\frac{\Delta, \Gamma, \sigma \vdash\ ?\ :\ T \qquad T \ \stackrel{\text{n}}{\twoheadrightarrow}\ (x : X) \to Y}{\Delta(\Gamma(x : X) \vdash\ ?_Y : Y), \Gamma, \sigma \vdash\ ? \rightsquigarrow (\lambda(x : X) \to ?_Y)\ :\ T} \begin{pmatrix}\text{abstraction}\\ \text{refinement}\end{pmatrix}$$

The side condition states that the target type normalizes to a function type.

If target type instead normalizes to a signature, the corresponding structure is a refinement.

$$\frac{\Delta, \Gamma, \sigma \vdash\ ?\ :\ T \qquad T \ \stackrel{\text{n}}{\twoheadrightarrow}\ \mathcal{S}}{\Delta', \Gamma, \sigma \vdash\ ? \rightsquigarrow \textbf{struct}\ \{x_1 = ?_1;\ \ldots;\ x_n = ?_n\}\ :\ T} \begin{pmatrix}\text{signature constr.}\\ \text{refinement}\end{pmatrix}$$

The symbol '$\mathcal{S}$' again refers to the generic signature. The new meta environment, Δ', is Δ extended with one new meta variable for each constituent of the structure.

$$\Delta' \ \equiv\ \Delta(\Gamma \vdash ?_1 : X'_1, \ldots,\ \Gamma \vdash ?_n : X'_n)$$

The target types of the subproblems, X'_k, are equal to X_k unless there are type dependencies between the signature elements. In that case the appropriate meta variables must be substituted into the types. Generally, the following substitutions are carried out.

$$X'_k \equiv X_k[?_{k-1}/x_{k-1}]\ldots[?_2/x_2][?_1/x_1]$$

The refinement rule for construction of data type elements is similar and is therefore left out.

In case there are type dependencies, meta variables will enter the target types. Later on they may also appear in the variable environment, as a consequence of further λ-abstractions. This is why unification of types is required and also why refinements and sub solutions are tagged with meta variable constraints.

4.2 Elimination Rules

In the case of introduction rules, the search is well guided by the target type. For elimination rules the situation is different. If you would simply use the elimination rules of the language, they would be valid for any target type. This would lead to a highly blind search of infinite depth. To avoid this, we demand that there is an appropriate hypothesis in the environment before producing an elimination refinement. But a type can be compound on more than one level, so in order to find e.g. C in $A \to (B \wedge C) \vee D$, the hypotheses must be completely taken apart within the same refinement. We do not have to do this for every subproblem. When a hypothesis is added to the environment, all possible decompositions can be computed. These decompositions will be called *elimination judgements*.

The elimination judgements have the following general form:

$$\Delta, \Gamma, \sigma \vdash M : U$$

It means that given the meta environment Δ, the variable environment Γ and value constraints σ, the term M is of type U.

Elimination judgements are generated from a number of *elimination inference rules* corresponding to the typing rules of the language.

First we have the initial and trivial rule that every variable can be used as it is.

$$\frac{\Gamma \vdash x : X}{\Delta, \Gamma \vdash x : X}\ \text{(reference)}$$

The occurrence of x below the line should be interpreted as the term referring to x. This rule is used as the starting point for every elimination judgement. All the other elimination inference rules presuppose a prior elimination judgement. Chains of judgements are thus constructed to iteratively decompose the hypotheses.

Now follow the inference rules containing actual elimination, one for each type class. First, we have application:

$$\frac{\Delta, \Gamma, \sigma \vdash M : U \qquad U \overset{\text{n}}{\twoheadrightarrow} (x : X) \to Y}{\Delta(\Gamma \vdash ?_X : X), \Gamma, \sigma \vdash (M\ ?_X) : Y[?_X/x]}\ \text{(application)}$$

The side condition is that U normalizes to a function type. Function invoking extends Δ with a meta variable corresponding to the argument of the application. For type dependent functions, the new meta variable will appear in the type of the elimination.

Next, if the type of the current elimination judgement normalizes to a signature, projection can be used to generate a new judgement for each element.

$$\frac{\Delta, \Gamma, \sigma \vdash \ M \ : \ U \qquad U \ \overset{\text{n}}{\twoheadrightarrow} \ \mathcal{S}}{\Delta, \Gamma, \sigma \vdash \ M.x_k \ : \ X'_k} \text{ (projection)}$$

where k is between 1 and n. The types of the new judgements must be modified so that all references to other elements are correctly qualified.

$$X'_k \ \equiv \ X_k[s.x_{k-1}/x_{k-1}] \ldots [s.x_2/x_2][s.x_1/x_1]$$

We must also be able to do data type elimination using case-expressions. Data type elimination introduces a new problem, since there is no direct connection between the decomposed types and the target type. As a consequence, it is difficult to know at which point the elimination involving a certain data type object should appear and if it should take place at all. The approach chosen here is to search the parts of every construction for a data type just as if they are all accessible. If a match is encountered, a refinement with a value constraint is generated. The idea is that solutions with complementary value constraints will later on be combined to form one solution with no constraints.

The data type elimination inference rule looks like this:

$$\frac{\Delta, \Gamma, \sigma \vdash \ M \ : \ U \qquad U \ \overset{\text{n}}{\twoheadrightarrow} \ \mathcal{D}}{\Delta, \Gamma, \sigma(M = \mathsf{c}_j \ x_{j,1} \ \ldots \ x_{j,n_j}) \vdash \ x_{jk} \ : \ X_{jk}} \text{ (data type elimination)}$$

where $j \in [1, m]$ and $k \in [1, n_j]$. There are $\sum_j n_j$ possible elimination judgements, one for each component of the data type. For each judgement, the set of constraints is extended with the constraint that M has the form of a certain construction. Note that the term of the new judgement refers to a variable that only appears in the value constraints. The constraints thus in a sense serve as an extra variable environment.

We now formulate the general elimination refinement rule:

$$\frac{\begin{array}{c}\Delta \subseteq \Delta', \quad \sigma \subseteq \sigma' \\ \Delta', \Gamma, \sigma' \vdash \ M \ : \ U \\ \Delta, \Gamma, \sigma \vdash \ ? \ : \ T \quad \Delta', \rho, \Gamma \vdash \ T \ \overset{\text{s}}{=} \ U\end{array}}{\Delta', \rho, \Gamma, \sigma' \vdash \ ? \rightsquigarrow M \ : \ T} \begin{pmatrix}\text{elimination} \\ \text{refinement}\end{pmatrix}$$

There are four side conditions. The two at the top demand that Δ' is an extension of Δ and σ' an extension of σ. This is guaranteed due to the constitution of the elimination inference rules. The third side condition is the elimination judgement saying that M is of type U. The last one assures that the target type can be

unified with U. An elimination judgement can include extension of both the meta environment, Δ, and the set of value constraints, σ. The extensions are reflected in the resulting refinement. The unification may produce meta variable constraints, ρ, which are also added to the refinement.

There is one situation which is not covered by the main elimination refinement rule. When there is a member of the empty type, i.e. absurdity, in the environment, it can be used to prove anything. A special elimination refinement rule is required. The empty type is in Agda represented by a data type with no constructors. In accordance with the typing rules, a case expression with no branches matches any type. Instead of comparing the type of the elimination judgement to the target type, the refinement rule for absurdity elimination demands that it normalizes to the empty type.

When the target type is an equality, the implementation of the elimination refinement rule behaves differently. Some knowledge about reflexivity, symmetry, transitivity and substitutivity has been hard-coded in the program. Equalities are proved and rewritten using these properties.

4.3 Case Analysis and Induction

The previous subsection described briefly how case-expressions for hypotheses are generated. But proofs may also contain case analysis on parameters. This is covered by a special refinement rule.

The case refinement rule must have some source which provides it with the collection of scrutinees to try. One place to look is in the environment. All objects that are parameters, are of data type and have not earlier been analyzed could qualify. Another way to generate scrutinees is to look at the target type and the environment elements' types and see which ones halt the computation, i.e. which scrutinees cannot be reduced to a construction and thus stop the type from being reduced any further. This seems somewhat more cunning than the first alternative, because it actually looks at the structure of the types involved. Another important advantage is that this method not only presents scrutinees that are single parameters, but also compound expressions, such as function applications where the argument is unknown. Case analysis on compound scrutinees is sometimes necessary. There is a special version of the case refinement rule dedicated to this. It is described further on in this subsection.

The implementation currently uses both these methods to produce candidates. The second does produce most relevant scrutinees, including compound ones. But it sometimes leaves out parameters that have a passive role in the types but still needs to be analyzed at that certain point in the proof.

For each candidate scrutinee which is a parameter, the refinement method generates a case expression with branches for each constructor of the data type. It also adds a locally defined function just outside the case expression. This is intended for recursive calls from within the branches or, in other words, for reference to the induction hypotheses of the problem. As arguments, the function takes the parameter itself, but to be flexible enough it must also take all objects in the environment whose type depends on the parameter. Furthermore, any

other parameter present in those objects' types should also be included and so forth.

In the subproblems of the refinement, the target type and variable environment are specialized for each branch by substituting the parameter with the correct construction and then re-simplifying the types. The locally defined function is added to the environment. It is however treated in a special way since we want to avoid bad recursive behaviour. The program currently restricts the use of the function in a way which only allows structural recursion, which guarantees program termination. This limitation is imposed by indicating that when the elimination judgements are generated for the function, it is not itself used as the starting point. Instead, the function applied to any recursive part of the parameter serves a starting points.

The rule for case refinement looks like this:

$$\frac{\Delta, \Gamma, \sigma \vdash ? \ : \ T \qquad \Gamma \vdash \ y : Y \qquad Y \overset{\mathrm{n}}{\twoheadrightarrow} \mathcal{D}}{\Delta', \Gamma, \sigma \vdash ? \rightsquigarrow \left(\begin{array}{l} \textbf{let } f \ (y' : Y) \ (\vec{z'} : \vec{Z}) : T = \\ \quad \textbf{case } y' \ \{ \\ \quad\quad \mathsf{c}_1 \ x_{1,1} \ \ldots \ x_{1,n_1} \rightarrow ?_1; \\ \quad\quad \vdots \\ \quad\quad \mathsf{c}_m \ x_{m,1} \ \ldots \ x_{m,n_m} \rightarrow ?_m \\ \quad \} \\ \textbf{in } \ f \ y \ \vec{z} \end{array} \right) : T} \text{ (case refinement)}$$

The side conditions of the rule are that y, the main parameter, is of type Y and that it in turn normalizes to a data type. The declarations $\vec{z} : \vec{Z}$ is the collection of secondary parameters and dependent objects. The meta environment is extended by one new meta variable for each branch of the case expression.

$$\Delta' \ \equiv \ \Delta(\Gamma'_1 \vdash ?_1 : T'_1, \ldots, \Gamma'_m \vdash ?_m : T'_m)$$

The variable environments and target types of these meta variable are the following:

$$\Gamma'_j \ \equiv \ \Gamma((f : (y' : Y) \rightarrow \vec{Z} \rightarrow T) \ (y' : Y) \ (\vec{z'} : \vec{Z}'_j) \ (x_{j,1} : X_{j,1}) \ldots (x_{j,n_j} : X_{j,n_j}))$$

$$T'_j \ \equiv \ T[\mathsf{c}_j \ x_{j,1} \ \ldots \ x_{j,n_j} / y]$$

$$\vec{Z}'_j \ \equiv \ \vec{Z}[\mathsf{c}_j \ x_{j,1} \ \ldots \ x_{j,n_j} / y]$$

In case the candidate scrutinee is a compound expression, the refinement is generated in a slightly different way. The side condition $\Gamma \vdash \ y : Y$ is replaced by $\Gamma \vdash \ M : Y$ where M is an arbitrary term and y' serves as a new parameter replacing M. Also, instead of substituting the variable y for the different constructions, every occurrence of the subexpression M is replaced. There is one more difference, namely that whenever possible, a proof of $M == y'$ is passed as an argument to the locally defined function. A proof of this equality is provided by reflexivity.

4.4 Generalization

Experiments with a method for introducing generalizations have been done. The method has two simple procedures for suggesting generalizations. The first looks for several occurrences of the same subexpression. If at least two occurrences are found, a generalization is constructed by replacing that subexpression with a new parameter of the correct type. The second procedure looks for two or more occurrences of the same parameter. Upon finding this, the parameter is separated into two or more new ones and all combinations of distributing the new parameters are tried.

The rule for generalization refinements has the following general form:

$$\frac{\begin{array}{c}\Gamma \vdash \vec{X} : *\\ \Gamma(\vec{x} : \vec{X}) \vdash T' : *\\ \Gamma \vdash \vec{M} : \vec{X}\end{array}\qquad \Delta, \Gamma, \sigma \vdash ? \ : \ T \qquad \Delta, \rho, \Gamma \vdash T \stackrel{s}{=} T'[\vec{M}/\vec{x}]}{\Delta', \rho, \Gamma, \sigma \vdash ? \rightsquigarrow \left(\begin{array}{l}\textbf{let } g\ (\vec{x} : \vec{X}) \ : T' = ?_g\\ \textbf{in } \ g\ \vec{M}\end{array}\right) \ : \ T} \left(\begin{array}{c}\text{generalization}\\ \text{refinement}\end{array}\right)$$

where

$$\Delta' \ \equiv \ \Delta(\Gamma(\vec{x} : \vec{X}) \vdash ?_g : T')$$

The variables $\vec{x}$ are the new parameters introduced by the generalization procedure and T' is the generalization of T. The rule has four side conditions. The first presents the types of the new parameters, $\vec{X}$. The second says that T' is a valid type in Γ extended by the new parameters. The third introduces the set of terms that will be the arguments of the function application. Finally, the last one ensures that by replacing $\vec{x}$ by $\vec{M}$ in T' the two types become equal.

5 Examples

A few examples are presented to illustrate what kind of problems can be solved by the tool. The examples refer to natural numbers and lists, which are defined as follows.

$$\begin{array}{l}\mathit{Nat} \ = \ \textbf{data } \mathtt{0} \mid \mathtt{s}\ (n : \mathit{Nat})\\ \mathit{List}\ X \ = \ \textbf{data } \mathtt{[]} \mid (::)\ (x : X)\ (\mathit{xs} : \mathit{List}\ X)\end{array}$$

Infix notation will be used for the constructor '::'.

5.1 Natural Numbers

As the first example, we look at commutativity for addition of natural numbers.

$$a + b == b + a$$

Addition for natural numbers is assumed to be defined as follows

$$\begin{array}{c}\mathtt{0} + b = b\\ \mathtt{s}\ a' + b = \mathtt{s}\ (a' + b)\end{array}$$

The tool solves this problem by induction on a. In the base case, induction on b is also done. In the inductive step, the target type is $\mathtt{s}\ (a' + b) == b + \mathtt{s}\ a'$. The hypothesis is used to replace $a' + b$ by $b + a'$, yielding $\mathtt{s}\ (b + a') == b + \mathtt{s}\ a'$. Now, an induction on b is done. The base case of this is proved by referring to reflexivity.

5.2 Lists

We take the following property of list reversing as an example:

$$rev\ (rev\ xs) == xs$$

The list reversing function is defined in terms of concatenation.

$$\begin{aligned} rev\ [] &= [] \\ rev\ (x :: xs') &= rev\ xs' +\!\!+ (x :: [])\end{aligned}$$

$$\begin{aligned} [] +\!\!+ ys &= ys \\ (x :: xs') +\!\!+ ys &= x :: (xs' +\!\!+ ys)\end{aligned}$$

The proof is by induction on xs. The ground case is trivial. The inductive step has the target type $rev\ (rev\ xs' +\!\!+ (x :: []))\ ==\ x :: xs'$ and the hypothesis ascertains $rev\ (rev\ xs') == xs'$.

The hypothesis is used to replace xs' with $rev\ (rev\ xs')$ in the RHS. This renders the problem $rev\ (rev\ xs' +\!\!+ (x :: []))\ ==\ x :: rev\ (rev\ xs')$.

Here, the subexpression $rev\ xs'$ appears twice and therefore the more general problem $rev\ (ys +\!\!+ (x :: []))\ ==\ x :: rev\ ys$ is attacked. The generalized problem is solved by induction on ys.

5.3 Quicksort

We prove the correctness of the quicksort algorithm.

$$\begin{aligned} qsort\ [] &= [] \\ qsort\ (x :: xs) &= qsort\ (filter\ (x >)\ xs) +\!\!+ (x :: qsort\ (filter\ (x \leq)\ xs))\end{aligned}$$

The type of the elements will be denoted by X, which is any ordered set. In order to adhere to structural recursion, we do the proof for a modification of the algorithm, $qsort'$. This takes two extra arguments; a natural number, n, which is the recursor and a proof, p, that the length of xs is at most n. The definition of $qsort'$ is left out.

To prove the correctness, we show that the output list is sorted and that it is a permutation of the input list. This is established for $qsort'$ in $prop\_qsort'$. As a corollary, the result is then brought to $qsort$ in $prop\_qsort$.

$$\begin{aligned} prop\_qsort' &: Sorted\ (qsort'\ n\ xs\ p) \wedge Perm\ xs\ (qsort'\ n\ xs\ p) \\ prop\_qsort &: Sorted\ (qsort\ xs) \wedge Perm\ xs\ (qsort\ xs)\end{aligned}$$

To show this, the problem is divided into a number of lemmas. Each of them can automatically be solved by the tool. Some of the lemmas depend on previous ones. Apart from the ones that are listed below, a couple of trivial properties of natural numbers and booleans have to be provided in a few cases. The main proposition, *prop_qsort'*, was then proved using several lemmas and *prop_qsort* is a trivial consequence.

Most of the lemmas are general properties for *Perm*, *Member* and *occs*, which counts the number of occurrences of an element in a list. In all propositions, non-proof arguments are omitted.

$$
\begin{aligned}
lem_1 &: Member\ x\ xs \leftrightarrow \neg(occs\ x\ xs == 0)\\
lem_2 &: Perm\ xs\ ys \rightarrow (x : X) \rightarrow (Member\ x\ xs \leftrightarrow Member\ x\ ys)\\
lem_3 &: occs\ x\ (xs{+\!+}\ ys) == occs\ x\ xs + occs\ x\ ys\\
lem_4 &: Perm\ (xs{+\!+}\ ys)\ (ys{+\!+}\ xs)\\
lem_5 &: Perm\ zs\ (xs{+\!+}\ ys) \rightarrow Perm\ (x :: zs)\ (xs{+\!+}\ (x :: ys))\\
lem_6 &: Perm\ xs_1\ xs_2 \rightarrow Perm\ ys_1\ ys_2 \rightarrow Perm\ (xs_1{+\!+}\ ys_1)\ (xs_2{+\!+}\ ys_2)\\
lem_7 &: Perm\ xs\ ys \rightarrow Perm\ ys\ zs \rightarrow Perm\ xs\ zs
\end{aligned}
$$

The remaining two lemmas are specific to the algorithm.

$$
\begin{aligned}
lem_8 &: Perm\ xs\ (filter\ (x >)\ xs{+\!+} filter\ (x \leq)\ xs)\\
lem_9 &: Sorted\ xs \rightarrow Sorted\ ys \rightarrow ((x : X) \rightarrow Member\ x\ xs \rightarrow |x \leq a|)\\
&\quad \rightarrow ((x : X) \rightarrow Member\ x\ ys \rightarrow |a \leq x|) \rightarrow Sorted\ (xs{+\!+}\ (a :: ys))
\end{aligned}
$$

Time cost of a search is measured by the number of refinements generated before a solution is found. The problems above typically take around 1000 refinements. The most difficult, lem_4 takes around 6000 refinements. These numbers are valid for searches where the needed lemmas are specified as hints.

6 Conclusions and Future Work

The experiments have resulted in a tool, that can construct proof terms for a fairly wide variety of problems. Inductive proofs can be constructed automatically, including nested induction. The tool can find some proofs in which a generalization is needed at some point to strengthen the induction.

The way types are compared and our approach for data type elimination substantially improve the efficiency of the search. The response time of the tool is in most cases satisfactory for an interactive proving system.

Among possible continuations of this work, we consider the following issues most interesting:

- Make the generalization more general. Investigating how far generalization can be automated is an interesting field.
- Make the induction capabilities more flexible by handling non-structural or mutual recursion. It would also be interesting to develop the mechanism that picks the parameter to do induction on, if possible.

– Further investigate the effects of using normal forms and syntactical comparison for types in type theory.
– Use testing (cf. e.g. [Hai03])to restrict search space. As the flexibility of the refinement methods grows, search complexity tends to increase. This should be compensated by improving the mechanism that limits the search forking. If e.g. generalization is improved, more candidates will be generated. Some of these may however be too general, i.e. false. It should be possible to disqualify some candidates by falsification. Testing could also be useful when invoking lemmas.

References

[CC99] C. Coquand and T. Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages*, Paris, France, Sep 1999.

[Coq98] Catharina Coquand. The AGDA Proof System Homepage. `http://www.cs.chalmers.se/~catarina/agda/`, 1998.

[Dow01] G. Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 16, pages 1009–1062. Elsevier Science, 2001.

[Hai03] Qiao Haiyan. *Testing and Proving in Dependent Type Theory.* PhD thesis, Department of Computing Science, Chalmers University of Technology, 2003.

[Hal00] Thomas Hallgren. Home Page of the Proof Editor Alfa. `http://www.cs.chalmers.se/~hallgren/Alfa/`, 1996-2000.

[HR00] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer, 2000.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[TS98] Tannel Tammet and Jan Smith. Optimized encodings of fragments of type theory in first-order logic. *JLC: Journal of Logic and Computation*, 8, 1998.

Cooperating Theorem Provers: A Case Study Combining HOL Light and CVC Lite

Sean McLaughlin and Clark Barrett[1]

Department of Computer Science, New York University
{seanmcl,barrett}@cs.nyu.edu

Abstract. HOL Light is a lightweight interactive theorem prover in the LCF style. CVC Lite is a fast, automated first order theorem prover that produces proofs of its deductions. This paper is a case study in combining theorem provers. We define a derived rule in HOL Light, `CVC_PROVE`, which, given a term of a theory supported by CVC Lite, calls the CVC Lite decision procedure and translates the solution back into HOL Light. This technique fundamentally expands the capabilities of HOL Light, in the sense that some valid terms that are intractable in the current HOL Light decision procedures become quickly provable. Furthermore, CVC Lite supports decision procedures for theories that do not exist in HOL Light. For instance, it decides unquantified statements in the theory of arrays. We give a minimal set of array definitions in HOL Light, and extend the translation mechanism. The result is that `CVC_PROVE` is able to prove difficult theorems about arrays with an extension of the HOL Light source of 8 lines of definitions and 6 basic proofs! After a brief historical discussion, we give the details of the translation mechanism. We give an example of a class of problems that were solved with `CVC_PROVE` but were unsolvable otherwise. Other less successful examples are given for comparison. We discuss the theory of arrays, and the minimal effort required to add an array theory to HOL Light with the help of CVC Lite. Finally, we discuss potential applications and future work.

1 Introduction and History

There are many different ways to formalize mathematics. Similarly, there are many kinds of problems that need to be solved in a rigorous way. As a result, there are a number of theorem provers being used today, all with differing underlying logics and all with different strengths and weaknesses. It is unfortunate that the efforts of the formal methods community are so ramified. For instance, the exciting advances made in TPS [5] are generally unavailable to a user of Mizar [16] without extensive study and programming. Users are loathe to change systems and even having a desired theorem proved in another system is often unsatisfactory. What is needed is some way to harness the power of another theorem prover without having to leave the environment of one's own system.

There are some efforts underway to address these issues. One is the nascent Logosphere project [1]. This promises to be a database of theorems in varying formats with a translation mechanism between the various logics.

In the hope of encouraging further progress along these lines, this paper describes a case study in which proofs from the automatic theorem prover CVC Lite are translated into corresponding proofs in the interactive theorem prover HOL Light.

1.1 HOL Light

HOL Light [11] is an interactive theorem prover descended from the LCF projects [8, 14] and the HOL4 theorem prover [2].

All the theorems are created by a core set of 10 primitive inference rules such as modus ponens and reflexivity. All other rules of inference are conservatively derived from these rules. The core system consists of those 10 rules and 3 logical axioms. With OCaml as the metalanguage, the user may program arbitrary new rules that cannot compromise the correctness of the system. This is ensured by defining an abstract type `thm` with no primitive constructor, and is enforced by the OCaml type system. The core consists of just over 300 lines of OCaml. HOL Light has been used extensively by its author to verify hardware designs at Intel [12]. But because of its transparent design and minimal base of trusted code, HOL Light was also chosen by Thomas Hales as the system in which to formalize his proof of the Kepler Conjecture (see Section 5.2). A large body of mathematics has been formalized in the system, from the construction of the real numbers to basic results in transfinite set theory and real and complex analysis.

1.2 CVC Lite

CVC Lite [6] is an automatic proof-producing theorem prover for decidable first order theories. It is derived from the SVC and CVC projects at Stanford University [7, 15]. It is one of the fastest theorem provers in existence today, solving problems in seconds that take hours for systems like HOL Light. The logical core differs in many ways from the HOL Light kernel. For example, as speed is a design goal of the system, there are many more primitive inference rules in CVC Lite. In fact, there are over one hundred rules alone for the theory of real linear arithmetic. (Contrast this number with the 10 total inference rules of HOL Light, where the reals are constructed from the axiom of infinity.) The trusted code base

is correspondingly larger, over 3000 lines being used to solve problems of linear real arithmetic. A natural question is whether we can access the speed and power of the CVC Lite engine without having to rely on its soundness.

2 Translation

There are numerous ways of connecting another prover like CVC Lite to HOL Light. One would be to accept theorems generated by CVC Lite as valid theorems. However, any bug in CVC Lite would compromise the soundness of HOL Light. A less intrusive approach is to *tag* theorems proved by CVC Lite [9]. This amounts to proving a theorem under the assumption **false**. While logically meaningless, this approach would allow the propagation of CVC Lite proofs so that, when faced with a theorem $CVC \vdash P$, we can say with certainty that if the output of CVC Lite was correct, then P holds. Fortunately, because CVC Lite can produce proofs, there is another alternative which is true to the spirit of HOL Light. This is to translate the proofs produced by CVC Lite into actual HOL Light proofs. We implement a HOL Light derived rule for each CVC Lite inference rule and translate the proof tree, calling the HOL Light rules as necessary from the bottom up, constructing the proof on traversal. Thus, a bug in CVC Lite would not compromise the HOL Light system. A false proof generated by CVC Lite would simply fail to translate into HOL Light.

2.1 Languages

HOL Light is written in the OCaml language [3]. CVC Lite is written in C++. The first challenge was getting the two languages to interact. A C interface for CVC Lite was written, allowing one to construct arbitrary CVC Lite formulae and to query the validity of a formula from a C program. OCaml functions were then written to call the C functions from the OCaml read-eval-print loop. The communication process is complicated by the different memory management systems of the two languages and imposes obvious limitations. Also, because it uses general strategies tailored for large proofs, even simple proofs in CVC Lite can be surprisingly large. A naive proof of $x + y = y + x$ runs some 15 lines. An example described below produces a multi-gigabyte proof. It is easy to produce proofs that exhaust main memory on a modern computer. With a bit more work, one can find a problem whose proof does not fit inside any

modern hard disk. A real concern for problems we are currently investigating is their ability to fit in the section of the heap allocated to C++ when the languages are combined. This was more of a software engineering feat than one of interest to the logic community. The details and code can be found on the Internet at [13].

2.2 Terms

After connecting OCaml to the C interface of CVC Lite, the next task was to translate terms between HOL Light and CVC Lite. Given that the CVC Lite logic is close to a subset of the HOL Light logic, this was relatively straightforward. The types of the terms we considered had obvious analogues. The term translation algorithm performs a depth first search of the term, constructing a term in the other system recursively.

Though it was not difficult for the part of CVC Lite we considered, it seems that such a translation may not always be so easy. Systematically translating between set theory, and typed lambda calculus, for instance, would take much deeper consideration.

2.3 Proofs

Translating proofs formed the heart of the research. As an illustration, we demonstrate a proof of the term 'x=x' in CVC Lite.

$$
\begin{array}{l}
(\textbf{iff–mp}\ \ \textbf{true}\ (=\ \ x\ x) \\
\quad (\textbf{proof–by–contradiction}\ \textbf{true} \\
\qquad (\textbf{let–p}\ ((\textbf{assump}_1\ (\textbf{not}\ \ \textbf{true}))) \\
\qquad\quad (\textbf{iff–mp}\ (\textbf{not}\ \ \textbf{true})\ \ \textbf{false}\ \ \textbf{assump}_1\ (\textbf{rewrite–not–true}))) \\
\quad (\textbf{iff-symm}\ (=\ x\ x)\ \ \textbf{true}\ (\textbf{rewrite–eq–refl}\, x))))
\end{array}
$$

Some examples of the rule semantics are:

- $(\textbf{iff–mp}\ \ [t_1]\ \ [t_2]\ \ [\vdash (t_1 = t_2)]\ \ [\vdash t_1]) \implies [\vdash t_2]$
- $(\textbf{proof–by–contradiction}\ \ [t]\ \ [\textbf{not}\ \ t \vdash \textbf{false}]) \implies [\vdash t]$
- $(\textbf{rewrite–not–true}) \implies \vdash \textbf{not}\ \ \textbf{true}\ =\ \ \textbf{false}$

In order to translate these rules to HOL Light, we wrote a derived rule for each CVC Lite rule encountered in the proof tree. Thus, to translate

proof–by–contradiction, we must define[1] a HOL Light derived rule where, given a proof of **false** from the term $\neg t$, a proof of t is produced.

```
let CCONTR =
  let P = `P:bool` in
  let pth = TAUT `(~P ==> F) ==> P` in
  fun tm th ->
    try let tm' = mk_neg tm in
        MP (INST [tm,P] pth) (DISCH tm' th)
    with Failure _ -> failwith "CCONTR";;
```

If the reader is unfamiliar with the HOL Light style, the crucial point is that we can define the rule in terms of previously defined rules of inference (here `MP, INST, TAUT, DISCH`). We have a similar rule for every inference rule in CVC Lite. We can then combine these rules in a recursive procedure that translates the proofs in a depth first traversal of the proof tree. The translation of the root proof node should yield the desired HOL Light theorem.

Note: It is interesting to consider the relative strengths of the inference rules in the two systems. For CVC Lite,

$$
\begin{aligned}
&(0 + 1y_4^2x_4^2 + 1y_4^2x_3^2 + 1y_4^2x_2^2 + -2y_4y_3x_2x_1 + \\
&\qquad 2y_4y_2x_3x_1 + -2y_4y_1x_4x_1 + 1y_3^2x_4^2 + 1y_3^2x_3^2 + \\
&\qquad 1y_3^2x_1^2 + 2y_3y_2x_3x_2 + -2y_3y_1x_4x_2 + 1y_2^2x_4^2 + \\
&\qquad 1y_2^2x_2^2 + 1y_2^2x_1^2 + 2y_2y_1x_4x_3 + 1y_1^2x_3^2 + \\
&\qquad 1y_1^2x_2^2 + 1y_1^2x_1^2 + 0 + 1y_4^2x_1^2 + 2y_4y_3x_2x_1 + \\
&\qquad -2y_4y_2x_3x_1 + 2y_4y_1x_4x_1 + 1y_3^2x_2^2 + -2y_3y_2x_3x_2 + \\
&\qquad 2y_3y_1x_4x_2 + 1y_2^2x_3^2 + -2y_2y_1x_4x_3 + 1y_1^2x_4^2) \\
&\qquad = \\
&(0 + 1y_4^2x_4^2 + 1y_4^2x_3^2 + 1y_4^2x_2^2 + 1y_4^2x_1^2 + \\
&\qquad 1y_3^2x_4^2 + 1y_3^2x_3^2 + 1y_3^2x_2^2 + 1y_3^2x_1^2 + \\
&\qquad 1y_2^2x_4^2 + 1y_2^2x_3^2 + 1y_2^2x_2^2 + 1y_2^2x_1^2 + \\
&\qquad 1y_1^2x_4^2 + 1y_1^2x_3^2 + 1y_1^2x_2^2 + 1y_1^2x_1^2)
\end{aligned}
$$

is a primitive inference. This corresponds to 101,359 HOL Light primitive inference rule applications!

[1] The rule CCONTR was written by John Harrison and is a part of the HOL-Light system. The actual rules we defined are longer and less instructive.

3 Results

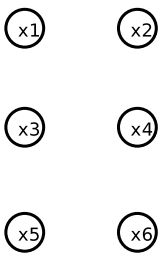
HOL Light and CVC Lite have two overlapping theories, those of real arithmetic and boolean satisfiability. These are the realms at which we aimed our translation mechanism in order to determine its relative effectiveness.

3.1 Satisfiability

Consider the following class of problems. You are given $n-1$ sets of n pigeon-holes, arranged in n rows of $n-1$ columns. Given that that no column can contain more than one pigeon, find a contradiction to the assertion that each row can contain a pigeon. For instance, this translates, for $n = 3$ as

$$\begin{aligned}&((\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \wedge \\ &(\neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_4) \wedge \\ &(\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6) \wedge \\ &(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6)) \rightarrow \textbf{false}\end{aligned}$$

for the picture



This is a notoriously difficult class of problems for typical boolean satisfiability methods. The following table[2] gives times for CVC Lite running alone (but still producing proofs), HOL Light running alone, and HOL Light using CVC Lite and performing the translation.

[2] All times are in seconds, running on a 1GH Pentium III running FreeBSD 5.2

n	CVC Lite	HOL Light	`CVC_PROVE`
2	0.10	4.5	1.75
3	0.18	13	10
4	0.90	34	43
5	2.9	*	210
6	19	*	980
7	238	*	4308

The empty entries under "HOL Light" are intractable in that system. Even the example with $n = 5$ ran for over 4 hours before we killed the process. We thus expand the power of HOL Light using the external system CVC Lite.

Note: The drastic slowdown between the CVC Lite program and the translation process requires some analysis. There are many places the inefficiency may reside. The OCaml process is running uncompiled under an interpreter. There is overhead from the many C function calls. There is also a great deal of inefficiency in the translation code. The lack of a profiler for the OCaml top level loop makes optimizing for HOL Light difficult. This could be overcome by packaging HOL Light and compiling the modules and in turn running the profiler. This option, along with other optimizations, will be explored in future research.

3.2 Real Arithmetic

The first problems we investigated with the translation process were terms of real linear arithmetic. The HOL Light decision procedure `REAL_ARITH` was unacceptably slow. This was a primary motivation for beginning the project in the first place. For instance, the following term

$$
\begin{aligned}
&(p_1^2 + q_1^2 + r_1^2 + s_1^2 + t_1^2 + u_1^2 + v_1^2 + w_1^2) \\
&\quad (p_2^2 + q_2^2 + r_2^2 + s_2^2 + t_2^2 + u_2^2 + v_2^2 + w_2^2) = \\
&(p_1p_2 - q_1q_2 - r_1r_2 - s_1s_2 - t_1t_2 - u_1u_2 - v_1v_2 - w_1w_2)^2 + \\
&\quad (p_1q_2 + q_1p_2 + r_1s_2 - s_1r_2 + t_1u_2 - u_1t_2 - v_1w_2 + w_1v_2)^2 + \\
&\quad (p_1r_2 - q_1s_2 + r_1p_2 + s_1q_2 + t_1v_2 + u_1w_2 - v_1t_2 - w_1u_2)^2 + \\
&\quad (p_1s_2 + q_1r_2 - r_1q_2 + s_1p_2 + t_1w_2 - u_1v_2 + v_1u_2 - w_1t_2)^2 + \\
&\quad (p_1t_2 - q_1u_2 - r_1v_2 - s_1w_2 + t_1p_2 + u_1q_2 + v_1r_2 + w_1s_2)^2 + \\
&\quad (p_1u_2 + q_1t_2 - r_1w_2 + s_1v_2 - t_1q_2 + u_1p_2 - v_1s_2 + w_1r_2)^2 + \\
&\quad (p_1v_2 + q_1w_2 + r_1t_2 - s_1u_2 - t_1r_2 + u_1s_2 + v_1p_2 - w_1q_2)^2 + \\
&\quad (p_1w_2 - q_1v_2 + r_1u_2 + s_1t_2 - t_1s_2 - u_1r_2 + v_1q_2 + w_1p_2)^2
\end{aligned}
$$

took over 1900 seconds with `REAL_ARITH`. In contrast, CVC Lite solves the problem in under 2 seconds. Using `CVC_PROVE`, we cut the time in half. We planned further optimizations until John Harrison, the author of HOL Light, produced a hyper-optimized version of `REAL_ARITH` that solved the problem in 21 seconds! With the new arithmetic procedure, `CVC_PROVE` is consistently around six times slower than `REAL_ARITH`.

The reasons for this are likewise numerous. The inferences made by CVC Lite were so large that the translation mechanism was forced to call `REAL_ARITH` itself just to prove the inferences were correct. Thus, we were forced to rely upon the very procedure we were trying to replace! We are currently designing a layer of inference rules in CVC Lite that more closely resemble the low level inference rules of HOL Light. In time we will see if such a translation process will be useful for pure linear arithmetic.

4 The Theory of Arrays

The experiments documented above arise from theories that exist in both theorem provers. A more obvious application of translation is to theories for which decision procedures do not yet exist in one of the provers. For instance, CVC Lite has a well developed theory of arrays. This theory does not exist in the current HOL Light version. As an alternative to implementing a decision procedure for arrays in HOL Light we extended the current translation mechanism to handle the CVC Lite array inference

rules. This gives us all the power of an array theory built in to HOL Light without the otherwise obligatory implementation effort.

4.1 Theory

The theory is a simple extensional theory of arrays, as found in [4]. Roughly, an array is a polymorphic type with two type variables, one corresponding to the indexing type, and the other corresponding to the value type. There are two constants, `read` and `write`. There are two axioms in the theory. One, the *axiom of extensionality* for arrays, saying that two arrays are equal if and only if they have the same elements. The second is a *read over write* axiom, giving a simple term reduction. The following is the entire contents of the HOL Light array theory:

```
new_type("array",2);; (* index_type, data_type *)

new_constant("read",`:(I,D)array->I->D`);;

new_constant("write",`:(I,D)array->I->D->(I,D)array`);;

let read_over_write = new_axiom(`!(a:(I,D)array) (i:I) (j:I) (v:D).
            ((i = j) ==> (read(write a i v) j = v)) /\
            (~(i = j) ==> (read(write a i v) j = (read a j)))`);;

let array_extensionality = new_axiom(`!(a:(I,D)array) (b:(I,D)array).
            (!(i:I). read a i = read b i) ==> (a = b)`);;
```

Note: Adding axioms to the HOL Light system is generally discouraged. A conservative extension theory of arrays is certainly possible, but the logic is greatly complicated. One way this could be accomplished is by defining an `(I,D)array` as the set of functions from type `I` to type `D` where a `read` is simply a function call and a `write` would be a conditional wrapper for the function

```
write A c v = (\x. if x = c then v else A x)
```

4.2 Results

Consider the following HOL Light term:

```
  ‘((S1:(real,real)array) = S2) ==>
  ((write S1 i (read S2 i)) = S1)‘;;
```

Given the axioms, the built-in HOL Light decision procedure can solve this problem in 56 seconds. `CVC_PROVE` takes .015 seconds.

Even slightly more difficult problems such as the following are intractable for HOL Light. By contrast, `CVC_PROVE` solved it in 7.6 seconds.

```
‘(((write (S1:(real,real)array) i v) = (write S2 j w)) /\
  (read S1 i = v) /\
  (read S2 j = w)) ==>
((S1 = S2) /\
 ((i = j) ==> (v = w)) /\
 (~(i = j) ==> (read S1 j = w)))‘;;
```

5 Future Research

We briefly consider some directions for future research.

5.1 Proof Size Reduction

There is an extensive proof theoretic literature on proof compaction. None of this is currently applied to the CVC Lite proofs. For larger problems, it could be necessary to translate proofs in pieces to allow the entire object to fit in memory. For instance, if the top level inference rule is **iff**–**mp**, one may create two separate processes to translate the separate parts of the proof, $\vdash (t_1 = t_2)$ and $\vdash t_1$. Once such a subproof exists as a HOL Light theorem, the proof can be deleted and another subproof begun. In this way it will be possible to handle proofs of practically limitless size.

5.2 The Flyspeck Project

The Flyspeck Project is an effort to formally verify Thomas Hales' 1998 proof of the Kepler Conjecture [10]. The proof relies critically on a large number of 6 and 7 dimensional real inequalities. These were proved using interval arithmetic and recursive branching using linear approximations and an explicit error bound given by Taylor's theorem. The verification of the inequalities required a large number of floating point arithmetic calls, often over 10^7 double precision multiplications in the course of a proof. These calculations are extremely slow in HOL Light. While still tractable in theory, a highly efficient implementation will be necessary to

prove them in practice. We are hoping that CVC Lite can be used as a tool to help guide the branching process to allow HOL Light to do as little work as possible in the proofs. In general, speed will be an issue on such a large project. We hope that CVC can help us attain the goal of verifying the proof in Hol Light.

6 Conclusion

This work demonstrates several benefits that can be derived from combining theorem provers. We have presented concrete examples of a qualitative increase in the power of HOL Light by translating proofs from CVC Lite. In a perfect world, any prover would be able to call any other for help in deciding terms automatically. This is a small step in that direction.

We'd like to thank New York University, the University of Pittsburgh, and the National Science Foundation for partial support of this work.

References

1. http://www.logosphere.org.
2. http://hol.sourceforge.net/.
3. http://www.ocaml.org/.
4. Clark W. Barrett Aaron Stump, David L. Dill and Jeremy Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *IEEE Symposium on Logic in Computer Science*, volume 16. IEEE Computer Society, 2001.
5. Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning. The TPS Theorem Proving System. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 641–642. Springer-Verlag, 1990.
6. Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *CAV*, 2004. To appear.
7. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity Checking for Combinations of Theories with Equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California.
8. M. Gordon, R. Milner, and C. P. Wadsworth. Edinburgh LCF: A Mechanised Logic of Computation. In *Lecture Notes in Computer Science*, volume 78. Springer-Verlag, 1979.
9. Elsa Gunter. Adding external decision procedures to HOL90 securely. In *Theorem Proving in Higher Order Logics 11th International Conference*, volume 1479 of *Lecture Notes in Computer Science*, pages 143–152. Springer-Verlag, 1998.
10. Thomas Hales. http://www.math.pitt.edu/thales/kepler98.
11. John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996. see http://www.cl.cam.ac.uk/users/jrh/hol-light.

12. John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.
13. Sean McLaughlin. http://www.cs.nyu.edu/seanmcl/cvc_hol_translation.
14. L. Paulson. Logic and Computation: Interactive Proof with Cambridge LCF. In *Cambridge Tracts in Theoretical Computer Science*, volume 2. Cambridge University Press, 1987.
15. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.
16. Andrzej Trybulec. http://mizar.uwb.edu.pl/.

Embedding Multiway Decision Graphs in HOL

Tarek Mhamdi and Sofiène Tahar

Concordia University
Department of Electrical and Computer Engineering
1455 de Maisonneuve West
Montreal, Quebec, H3G 1M8, Canada
{mhamdi,tahar}@ece.concordia.ca

Abstract. In this paper, we present an embedding of Multiway Decision Graphs (MDG) in the HOL theorem prover. We first embedded the MDG underlying logic and then implemented a set of MDG graph manipulation operators and algorithms. This platform allowed us to develop state-exploration based applications inside HOL, such as MDG reachability analysis, equivalence checking and model checking. Furthermore, we also developed decision procedures for equivalence and tautology checking based on the MDG tool. The proposed embedding provides a verification framework, in which the verification problem is specified in HOL, while the proof is derived by tightly combining the MDG based computations and the HOL theorem prover facilities.

1 Introduction

Whenever an error creeps into a design, time and money must be spent to locate the problem and correct it, and the longer a bug evades a detection, the harder and more expensive it is to fix. As design complexity increases, simulation times become prohibitive and coverage becomes poor, allowing numerous bugs to slip through to later stages of the design cycle. What is needed, therefore, is a complement to simulation for determining the correctness of a design. For this reason, there has been a surge of research interest in *formal verification* techniques [14]. In general, formal verification problem consists of mathematically establishing that an implementation satisfies a specification. The implementation refers to the system design that is to be verified and the specification refers to the property with respect to which the correctness is to be determined. Formal verification methods fall into two categories [12]: *proof-based* methods, mainly theorem proving and *state-exploration* methods, mainly model checking and equivalence checking. While theorem proving is a scalable technique that can handle large designs, model checking suffers from the so-called state-explosion problem which prevents its application to industrial systems [15]. On the other hand, while model checking is fully automatic, deriving proofs is a user guided technique that requires a lot of expertise and hence can be tedious and difficult.

Both techniques do not allow the automatic verification of large systems. So, various compromises are being explored to combine the strengths of both.

They can be summarized as : (i) tools integration, (ii) adding deduction rules to a state of the art checking tool or (iii) deeply embedding checking algorithms inside a theorem prover. For the first approach, we start with two stand-alone tools, a theorem prover and a checking tool, where we link the latter to the theorem prover using scripting languages to be able to automatically verify small sub-goals generated by the theorem prover from a large system. The starting point of the second approach is a state-of-the-art checker to which we add proving rules to hopefully extend the verification to complete systems. Finally, the third approach, which is the one we adopted in our work, consists of embedding algorithmic infrastructures inside a theorem prover resulting in a hybrid system tightly combining checking algorithms and proving facilities. This approach differs from the first one in the way the verification is performed. In fact, we do not use an external checking tool, instead we develop state-exploration algorithms inside the theorem prover.

In this work, we developed a platform of state-exploration algorithms inside the HOL proof system [9]. Our decision diagram data structure is the *Multiway Decision Graphs* (MDGs) [5], which we integrate in HOL as a built-in datatype. The logic underlying MDGs will be embedded as a theory that provides the tools to specify the verification problem in the logic supported by the MDGs. The specification will consist of a set of HOL formulae that can be represented by their correspondent MDGs. Operations over these formulae will be viewed as MDG operations over their respective graphs. An MDG package will, then, be used to build the graph representation of HOL formulae allowing the manipulation of graphs rather than HOL terms. Once available inside the theorem prover, the MDG data structure and operators can be used to automate parts of the verification problem or even to write state enumeration algorithms like reachability analysis or model checking.

The organization of this paper is as follows: Section 2 reviews some related work. Section 3 describes the embedding of the logic underlying the MDGs in HOL. Section 4 shows how HOL is linked to the MDG package. Section 5 describes the embedding of the reachability analysis procedure. Sections 6 and 7 illustrate the use of the embedding in the implementation of state-exploration algorithms. Finally, Section 8 concludes the paper and gives some future research directions.

2 Related Work

The quest for an efficient combination of theorem proving and model checking has long been one of the major challenges in the field of formal verification. The work described here has been strongly influenced by the HolBdd [6, 7] system developed by Gordon. HolBdd consists of a platform allowing the programming of Binary Decision Diagram, (BDD) [3] based symbolic algorithms in the Hol98 proof assistant. It provides intimate combinations of deduction and algorithmic verification. They use a small kernel of ML [10] functions to convert between

BDDs, terms and theorems. Their work was applied to perform reachability programming in Hol98.

A pioneering work in the area is the one of Joyce and Seger [11] combining HOL and the symbolic trajectory evaluation (STE) tool VOSS. HOL-VOSS presents a mathematical link between the specification language of the VOSS system and the specification language of HOL. A tactic, VOSS_TAC, was implemented as a remote function. It calls the VOSS system as a child process of the HOL system to check whether an assertion, expressed as a term of higher-order logic, is true. If this is the case, the assertion will be turned to a HOL theorem. The early experiment with HOL-VOSS suggested that a lighter theorem prover component was sufficient, since all that was needed was a way of combining results obtained from STE. A system based on this idea, called VossProver was developed. As a continuation of HOL-VOSS, Aagaard *et al.* [1] developed the Voss-ThmTac system combining the ThmTac theorem prover with the VOSS system. Its power comes from the very tight integration of the two provers, using a single language, FL, as both the theorem prover's meta-language and its object language.

Rajan *et al.* [20] described an approach where a BDD based model checker for the propositional μ-calculus has been used as a decision procedure within the framework of the PVS [18] proof checker. They used μ-calculus as a medium for communicating between PVS and the model checker. It was formalized by using the higher-order logic of PVS. The temporal operators are given the customary fixpoint definitions using the μ-calculus. These expressions were translated to the form required by the model checker. The latter was then used to verify the subgoals generated within PVS.

Schneider and Hoffmann [21] linked the SMV model checker [16] to HOL using PROSPER. They embedded the linear time temporal logic (LTL) in HOL and translated LTL formulae into equivalent ω-Automata, a form that can be reasoned about within SMV. The translation is completely implemented by means of HOL rules. On successful model checking, the results are returned to HOL and turned to theorems. The deep embedding of the SMV specification language in HOL allows LTL specifications to be manipulated in HOL.

In [13], [19] and later [17] a hybrid tool and a methodology tailored to perform hierarchical hardware verification have been developed by the *Hardware Verification Group* of *Concordia University*. They integrate the HOL theorem prover to the MDG equivalence checker and later to the MDG model checker. The work is done within the proof system but using the specification style of the automated verification tool. The HOL-MDG tool is used to verify that a structural specification of hardware implementation implies its behavioral specification. They perform the equivalence checking within the MDG tool by applying a HOL tactic MDG_EQ_TAC. This latter mainly generates the MDG required files and ensures the interaction with the MDG equivalence checker. If the design is large enough to cause state explosion, and since the description model is written in a hierarchical way, a tactic HIER_VERIF_TAC is called to break the design into sub-blocks. The same procedure is recursively applied if necessary. At any point,

the goal proof can be done in HOL. Similarly, they provide a way to express temporal properties inside the theorem prover and support the full properties specification language of MDG by introducing abstract datatypes and uninterpreted functions. A HOL tactic, called MDG_MC_TAC is used to perform model checking. It supports hierarchical verification and model reduction.

While [13, 17, 19] describe systems integrating two stand-alone tools, namely, HOL and an external MDG tool, the work described here is not intended to use an external tool to verify subgoals. Instead MDGs are a built-in datatype of HOL and operators over MDGs are available in the proof system which allows us to tightly combine HOL deduction and MDG computations. Besides, state-exploration algorithms will be written inside HOL. Thereafter, the main difference between our approach and the HOL-MDG tool is that our embedding provides a secure and general programming infrastructure to allow the users to implement their own MDG-based verification algorithms inside the HOL system.

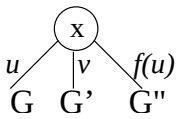
The work in [1, 11, 21] use the same approach as the HOL-MDG hybrid tool in the way they integrate the model checker to the theorem prover. The work in [20] uses the μ-calculus as a medium for communicating between the theorem prover and the model checker. It is a shallow embedding of stand-alone tools language while ours is a deep embedding of the decision diagram data structure and its operators are embedded inside the theorem prover.

Obviously, the most related work to ours is that of Gordon [6, 7]. Our work, however, deals with embedding MDGs rather than BDDs. In fact, while BDDs are widely used in state-exploration methods, they can only represent Boolean formulae. On the other hand, MDGs represent a subset of first-order terms allowing the abstract representation of data and hence raising the level of abstraction.

3 Embedding The MDG Logic in HOL

3.1 Multiway Decision Graphs

A *Multiway Decision Graph* (MDG) is a finite directed acyclic graph G where the leaf nodes are labeled by formulae, the internal nodes are labeled by terms, and the edges issuing from an internal node N are labeled by terms of the same sort as the label of N. Such a graph represents a formula defined inductively as follows: (i) if G consists of a single node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1, ..., B_n$ leading to subgraphs $G'_1, ..., G'_n$ and if each G'_i represents a formula P_i then G represents the formula $\vee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$. For example, if x, u, and v are variables of abstract sort α, f is a function symbol of type $\alpha \rightarrow \alpha$, and G, G', and G'' represent P, P', and P'', respectively, then the graph



represents the formula

$$((x = u) \wedge P) \vee ((x = v) \wedge P') \vee ((x = f(u)) \wedge P''). \quad (1)$$

The above is of course too general, a set of *well-formedness conditions* [5] turns MDGs into *canonical representations* that can be manipulated by efficient algorithms. More details on MDG are described in the sections to follow.

Multiway Decision Graphs are intended to represent Abstract State Machines (ASM) [5], an abstract description of state machines based on a many-sorted first order logic with a distinction between abstract and concrete sorts. More details on MDGs are described in the subsections to follow.

3.2 MDG Sorts

Concrete sorts have enumerations, while abstract sorts do not. An enumeration is a finite set of constants. This is embedded in HOL as follows:

– `Concrete_Sort = Concrete_Sort of string ⇒ string list;`

It declares a constructor called *Concrete_Sort* that takes as arguments a sort name and its enumeration to define a concrete sort. For example, if *state* is a concrete sort with [stop, run] as enumeration, then this is declared in HOL by:

```
val state = Define `state = Concrete_Sort ``state`` [ stop; run ]`;
```

– `Abstract_Sort = Abstract_Sort of 'a;`

To define an abstract sort of type *alpha* (which means that the sort is actually abstract and hence can represent any HOL type) we use the *Abstract_Sort* constructor as follows:
val *alpha* = Define `*alpha* = Abstract_Sort "*alpha*"`;
To determine whether a sort is concrete or abstract, we use predicates over the sorts constructors called *IsConcreteSort* and *IsAbstractSort*, where "_" means "don't care".

```
(IsConcreteSort (Concrete_Sort _ _)  = T) /\ (IsConcreteSort _ = F);
(IsAbstractSort (Abstract_Sort _)    = T) /\ (IsAbstractSort _ = F);
```

These predicates will be used for instance to determine the sort of a variable or a function symbol.

The vocabulary consists of concrete and generic constants, variables and function symbols (also called operators). The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an *abstract function symbol.* Abstract function symbols are used to denote data operations and are uninterpreted. If all $\alpha_1 ... \alpha_{n+1}$ are concrete, f is a *concrete function symbol.* Concrete function symbols, and concrete constants as a special case, can always be entirely interpreted and thus be eliminated; for simplicity, we assume that they are not used. Finally, if α_{n+1} is concrete while at least one of $\alpha_1 ... \alpha_n$ is abstract, then we refer to f as a *cross-operator.*

3.3 MDG Variables

An abstract variable can be either primary or a secondary variable. A primary variable labels a node in the graph while a secondary variable is an abstract variable occurring in the argument list of a function symbol. It can also be an abstract variable labeling an edge in the graph. In our embedding, a primary abstract variable will be declared using the *Abstract_Var* constructor while a secondary variable will be declared using the *Secondary_Var* constructor.

– `Concrete_Var = Concrete_Var of string ⇒ Concrete_Sort;`

A variable is specified by its name and sort. A concrete variable is a variable of concrete sort. For example, If x is a variable of sort *state*, declared above, then this is written in HOL as follows:

```
val x = Define `x = Concrete_Var ``x`` state`;
```

– `Abstract_Var = Abstract_Var of string ⇒ Abstract_Sort;`

An abstract variable y with name "y" and sort *alpha* is declared using:

```
val y = Define `y = Abstract_Var ``y`` alpha`;
```

– `Secondary_Var = Secondary_Var of string ⇒ Abstract_Sort;`

The *Secondary_Var* constructor is similar to the *Abstract_Var* constructor. For example:

```
val y1 = Define `y1 = Secondary_Var ``y1`` alpha`.
```

In this case also, we use some predicates to determine whether a variable is concrete, abstract or secondary. They are called, respectively, *IsConcreteVar*, *IsAbstractVar* and *IsSecondaryVar*.

3.4 MDG Constants

A constant can be either an individual (concrete) constant or an abstract generic constant. The latter is identified by its name and its abstract sort. The individual constants can have multiple sorts depending on the enumeration of the sort in which they are. In HOL they are declared as follows:

– `Individual_Const = Individual_Const of string;`

The enumeration of the concrete sort *state* is "[stop , run]". *stop* and *run* are two individual constants that have *state* as their sort. They must be defined in order to be able to declare the sort state.

```
val stop = Define `stop = Individual_Const ``stop```;
val run = Define `run = Individual_Const ``run```;
```

– `Generic_Const = Generic_Const of string ⇒ Abstract_Sort;`

Having declared "*alpha*" as abstract sort, we can declare generic constants of that sort. Say a is a generic constant of sort *alpha*.

```
val a = Define `a = Generic_Const ``a'' alpha`;
```

To check whether a constant is an individual constant or an abstract generic constant, we define two predicates, *IsIndividualConstant* and *IsGenericConstant*.

3.5 MDG Functions

MDG functions can be either concrete, abstract or cross-operators. As mentioned before, concrete functions are not used since they can be eliminated by case splitting. Cross-functions are those that have at least one abstract argument. But when we focus on terms that are concretely reduced, all the sub-terms of a compound term (abstract/cross function) have to be abstract. In addition they are secondary variables.

- `Cross_Function = Cross_Function of string ⇒ Secondary_Var list ⇒ Concrete_Sort;`

In general, a function is identified by its name, the sorts of its arguments and its sort. In this case, we specify the variables rather than sorts because we focus on cross-terms or abstract terms instead of the correspondent symbols. If *equal* is a function that checks if two abstract variables are equal, then, *equal* is a cross-function.

```
val bool  = Define `bool = Concrete_Sort "bool" ["0";"1"]`;
val y1    = Define `y1 = Secondary_Var ``y1'' alpha`;
val y2    = Define `y2 = Secondary_Var ``y2'' alpha`;
val equal = Define `equal = Cross_Function "equal" [y1;y2] bool`;
```

- `Abstract_Function=Abstract_Function of string ⇒ Secondary_Var list ⇒ Abstract_Sort;`

If *max* is a function that takes two abstract variables as arguments and returns the greater one, then *max* is an abstract function.

```
val max = Define `max = Abstract_Function ``max'' [y1;y2] alpha`;
```

The predicates *IsAbstractFunction* and *IsCrossFunction* are used to determine the nature of a compound term.

3.6 MDG Terms

MDG terms are either individual constants, generic constants, concrete or abstract variables, cross-operators or abstract function symbols. We provide a constructor called *MDG_Term* that is used every time a new term is declared. The single constructor is used so that terms will have the same type and hence can be used in equalities. In fact if x is declared using the *Concrete_Var* constructor and *stop* using the *Individual_Const* constructor, we will not be able to write an equation of the form $x = stop$ due to type mismatching. However, such an equation is possible if both are declared using the same constructor.

```
Hol_datatype `MDG_Term =
  Individual_Const  of string => Concrete_Sort
| Generic_Const     of string => 'a Abstract_Sort
| Concrete_Var      of string =>    Concrete_Sort
| Abstract_Var      of string => 'a Abstract_Sort
| Cross_Function    of string=>('a Secondary_Var)list=> Concrete_Sort
| Abstract_Function of string=>('a Secondary_Var)list=>'a Abstract_Sort`
```

3.7 Well-formed MDG Terms

For BDDs to be canonical, they have to be reduced and ordered. Similarly, MDG require certain well-formedness conditions to represent canonically the MDG terms. The set of well-formed terms that can be represented canonically by the MDGs is called the set of *Directed Formulae* (DF). Given two disjoint sets of variables U and V, a DF of type $U \rightarrow V$ is a formula in disjunctive normal form (DNF) such that

1. Each disjunct is a conjunction of equations of the form:
   - $A = a$, where A is a cross-term of concrete sort α containing no variables other than elements of U, and a is an individual constant in the enumeration of α, or
   - $u = a$, where $u \in U$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   - $v = a$, where $v \in V$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   - $v = A$, where $v \in V$ is a variable of abstract sort α and A is a term of type α containing no variables other than elements of U;
2. In each disjunct, the left hand sides of the equations are pairwise distinct; and
3. In each disjunct, every variable $v \in V$ should appear as the left hand side of an equation $v = A$.

Conditions 2 and 3 must be respected by the user when specifying the verification problem. The condition 3 is less stringent than it seems. In practice, one can introduce an additional dependent variable u and add an equation $v = u$ to a disjunct where an abstract v is missing.

For example, condition 1 is embedded in HOL and checked using the function *Well_formedTerm* that, recursively, calls *Well_formedEQ* to check the well-formedness of an equation.

```
fun Well_formedEQ eq =
            ((IsConcreteVar lhs)   /\ (IsConcreteConstant rhs)) \/
            ((IsCrossFunction lhs) /\ (IsConcreteConstant rhs)) \/
            ((IsAbstractVar lhs )  /\ (IsAbstractFunction rhs)) \/
            ((IsAbstractVar lhs )  /\ (IsAbstractVar rhs))      \/
            ((IsAbstractVar lhs )  /\ (IsGenericC rhs))         \/
             (IsBool lhs);
```

4 Linking HOL to The MDG Package

The MDG logic is embedded in HOL to make it possible to specify a verification problem in HOL in terms of formulae that can be represented by canonical MDGs. The next step would be to provide the tools to build and manipulate the graph representations of these formulae. This platform will consist of ML functions that call an MDG package[1] as an external process. The package is invoked using a script file, in which, the different manipulations to be done in MDG are specified. For example, to perform the conjunction of a list of well-formed Terms, we use the ML function *Conj*. This function calls an intermediate function to write the script file corresponding to a conjunction, then calls the specific MDG functions to perform the operation and eventually return the result to HOL. The ML functions pass the script file to the MDG package using the *system* function [10]. The latter computes the result (MDG graph) and then writes it in a file "*mdghol.ch*". Using the function *ReadMdgOutput*, the result is retrieved.

4.1 Constructing MDGs in HOL

To construct the graph representation of a HOL term we use the function *termToMdg*. Well-formedness conditions are first checked using the predicate *Well_formedTerm*. It either raises an exception when this is not the case or begins gathering the information to call the package.

The first step is to determine the sorts of all the sub-terms using the function *ToMdgSorts*. If a sub-term is of concrete sort *Sort*, it is declared as "*concrete_sort(Sort,Enum)*", where *Enum* is the enumeration of *Sort*. When an abstract sort, say *alpha*, is encountered, then it is declared by "*abs_sort(alpha)*". For example, if a term A includes a concrete variable of sort *bool* and an abstract variable of sort *alpha*, then *ToMdgSorts* returns the following list:

$$[\text{"}conc\_sort(bool,[0,1])\text{."},\text{"}abs\_sort(alpha)\text{."}].$$

The second step is to declare all the variables, functions and generic constants used in the term. A variable is declared by "*signal(label,sort)*". A generic constant is declared by "*gen_const(label,sort)*". When a function is encountered, both the secondary variables and the function symbol must be declared. The function symbol is declared as "*function(f,[sorts],sort)*". *sorts* are the sorts of the secondary variables, arguments to the function symbol f. *sort* is its target sort.

Thereafter, *termToMdg* writes the variables order list in the script file and then calls the function *header* responsible for retrieving the list of the LHSs and RHSs of the equations in the term which will be the parameters of the *mdg* function. The latter is then called and the result is retrieved using the *readMDGOutput* function. Instead of returning the whole graph structure, we return only its ID which will be used to map the term to its MDG representation.

[1] We provide a lifted version of the MDG package with which we are able to call internal MDG functions.

4.2 Embedding MDG Basic Operators

The MDG operators are embedded, as well, to allow the manipulation of graphs rather than terms. we show below the basic MDG operators.

- *Conj* : performs the conjunction of a set of graphs;
- *Disj* : performs the disjunction of a set of graphs;
- *Relp* (Relational Product) : used for image computation. It takes the conjunction of a collection of MDGs, having pairwise disjoint sets of abstract primary variables, and, existentially quantifies with respect to a set of variables, either abstract or concrete, that have primary occurrences in at least one of the graphs. In addition, it can rename some of the remaining primary variables according to a renaming substitution;
- *PbyS* (Pruning By Subsumption) : used to approximate the set difference operation. Informally, it removes all the paths of a graph P from another graph Q.

5 Reachability Analysis in HOL

The reachability analysis is embedded using the MDG operators interfaced to HOL. We show here the different steps to compute the set of the reachable states of an abstract state machine.

5.1 Computing Next States

Let I, B and R be, respectively, a set of inputs, a set of initial states of a machine and its transition relation. The ML function *ComputeNext* representing the set of next states, computed from B with respect to R, is defined by:

$$\mathit{ComputeNext}(G_I\ G_B\ G_R) = \mathit{RelP}(G_I\ G_B\ G_R\ Q\ \eta).$$

where, G_I, G_B and G_R are the MDG representations for I, B and R, respectively. Q is the set of input and state variables over which the MDG is quantified. η is the renaming substitution. B can be the set of initial states as well as the set of states already reached by the machine.

5.2 Computing Outputs

The set of outputs corresponding to a set of initial states and inputs, with respect to an output relation O, is represented by the ML function *ComputeOutputs* below, where G_O is the MDG representation of O.

$$\mathit{ComputeOutputs}(G_I\ G_B\ G_O) = \mathit{RelP}(G_I\ G_B\ G_O\ Q)\ \text{“\_”}.$$

For every state of the machine, and a set of data inputs, corresponds a set of output values. These will be used to check an invariant.

5.3 Computing Frontier Set

The frontier set is the set of newly visited states. If V represents the set of states already visited, $V_n = \mathit{ComputeNext}(G_I\ V\ G_R)$ is the set of next states reached from V. In this case the frontier set is $V_n \setminus V$ which is represented by the ML function *ComputeFrontier*.

$$\mathit{ComputeFrontier}(V_n\ V) = \mathit{PbyS}(V_n\ V).$$

The frontier set is used to check if all the states reachable by the machine are already reached. If this is the case (the frontier set is empty), then the reachability analysis terminates and the set of reachable states is returned. If the frontier set is not empty, then new states were visited during the last iteration. In this case, the analysis continues until reaching the fixpoint (set).

5.4 Computing Reachable States

The set of reachable states is the set of all the states of a machine, starting from an initial state, for a certain set of inputs. For abstract state machines, the state space can be infinite. Hence, the set of reachable states may not exist[2]. Using the solutions proposed in [2], the set of reachable states is computed and represented by the function, *ComputeReachable*, defined by[3]:

ComputeReachable $G_I\ G_B\ G_R$ =
    $K = 0$, $S = G_B$
    loop
        $K = K + 1$
        $N =$ **ComputeNext** $G_{IK}\ G_B\ G_R$
        if **ComputeFrontier** $N\ S = F$ then return success
        $G_B =$ **ComputeFrontier** $N\ S$
        $S =$ **Disj** $N\ S$
    end loop
end;

ComputeReachable computes the set of reachable states S of a state machine described by its transition relation, starting from an initial state and for a certain data input. S is initialized to B (the initial state), and the sets of next-states are computed until reaching a fixpoint characterized by an empty frontier set.

6 Invariant and Model Checking in HOL

6.1 Invariant Checking

Invariant checking is a direct application of the reachability analysis algorithm. It consists of checking that a property or an invariant holds on the outputs

[2] This is called the non-termination problem which was tackled in [2] using various heuristics.

[3] For the sake of clarity, this is just a simplified version of the algorithm

of a state machine in every reachable state. First, the invariant is checked in the initial state. This is done by computing the outputs corresponding to that state and then using the MDG operators to check that these outputs satisfy the invariant. After that, next-states are computed and for every state reached, the invariant is checked on the outputs. In a given iteration, if the outputs of the machine satisfy the invariant, then the procedure continues for the next-state. If, on the other hand, the invariant does not hold, the analysis terminates and a failure is reported. A counterexample can be generated to trace the error. The invariant checking algorithm is implemented in HOL as an ML function *InvariantChecking* which takes as arguments:

- T_R: the transition relation specified as a list of directed formulae;
- O_R: the output relation specified by a directed formula;
- I_N: the initial state specified by a directed formula;
- *Inputs*: the input variables list;
- *States*: the state variables list;
- *NxStates*: the next-state variables list corresponding to *States*.
- *Inv*: the invariant to be checked specified as a directed formula.

The function *InvariantChecking*, first, builds the graphs of the transition relation, output relation, the initial state and the invariant using the function *termToMdg*. Then, generates the input graph. After that, the outputs are computed using *NewOutputs* and then the invariant is checked. If the invariant holds, the next-state variables are computed using *ComputeNext*. Checking the frontier set will cause the termination of the analysis or another iteration.

InvariantChecking T_R O_R I_N *Inputs States NxStates Inv* =
    // builds the MDG representations
    // generates the renaming substitution function
    $K = 0$, $S = G_{IN}$, $R = G_{IN}$
    loop
        $K = K + 1$
        // generates the input graph G_{IK}
        $O_S =$ **ComputeOutputs** G_{OR} R G_{IK}
        if (**PbyS** O_S G_{Inv}) $\neq F$ return failure
        $N =$ **ComputeNext** G_{IK} R G_{TR}
        if **ComputeFrontier** N $S = F$ then return success
        $R =$ **ComputeFrontier** N S
        $S =$ **Disj** N S
    end loop
end InvariantChecking;

6.2 Model Checking

Similarly, MDG temporal operators can be implemented in HOL for model checking. In the following we present how the operator **AF** on a first-order property

formula P [22] is embedded.

Check_AF T_R I_N *Inputs States NxStates* P =
    // builds the MDG representations G_{TR}, G_{IN}, G_P
    // generates the renaming substitution function
    $K = 0$, $\Sigma = F$, $C = G_{IN}$
    // Σ contains sets of states not satisfying P
    loop
        Q = **ComputeFrontier** C G_P
        // removes states satisfying P
        if $Q = F$ then return success
        if **ComputeFrontier** Σ $Q \neq \Sigma$ then return failure
        Σ = **Disj** Σ Q
        $K = K + 1$
        C = **ComputeNext** G_{IN} Q G_{TR}
    end loop
end Check_AF;

7 MDG as a Decision Procedure

The multiway decision graphs are a canonical representation of the directed formulae. Two directed formulae are equivalent if and only if they are represented by the same graph for a fixed order. This property can be used to prove automatically the equivalence of HOL terms or to check that a formula is a tautology in case it is represented by the MDG *true*.

7.1 Combinational Equivalence Checking

We provide here a decision procedure that enables us to verify automatically the equivalence of a certain subset of first-order HOL terms. This is performed using the ML function *EquivCheck*.

```
fun EquivCheck order t1 t2 =
    let   val s1 = termToMdg order t1
          val s2 = termToMdg order t2
    in
         (s1=s2)
    end;
```

Using *EquivCheck* we write an oracle that builds a theorem stating the equivalence between terms. The theorem is not derived from axioms and inference rules which will endanger the security provided by the HOL reasoning style. Theorems created using the oracle are tagged so that an error can be traced whenever it occurs. This kind of decision procedures are widely used to introduce some automation to the theorem provers.

7.2 Tautology Checking

A formula is a tautology if it is represented by the MDG T. This makes the check very easy for the subset we consider which are the directed formulae. We use the ML function *Tautology*.

```
fun Tautology order t =
    let val s = termToMdg order t
    in
        isTrue s
    end;
```

8 Conclusions and Future Work

In this paper, we proposed an approach that allows certain verification problems, specified in the HOL theorem prover, to be verified totally or in part using state-exploration algorithms. Our approach consists of an infrastructure of decision diagrams data structure and operators made available in HOL, which will allow the user to develop his own state-exploration algorithms in the HOL proof system. The data structure we considered in our work is the multiway decision graphs (MDG). MDG is an extension to the well-known binary decision diagrams in that it eliminates the state explosion problem introduced by the datapath.

The MDGs are embedded in HOL as a built-in datatype. Operations over the MDGs are interfaced to HOL functions allowing the manipulation of graphs rather then their correspondent HOL terms. Using the embedding of the logic underlying the multiway decision graphs in HOL, the verification problem is specified as a set of well-formed directed formulae that can be represented canonically by well-formed MDGs. This is made possible thanks to the lifted MDG package that we provided and interfaced to HOL resulting in a platform of functions to represent terms by their correspondent MDGs and manipulate them.

The platform, we provide, allowed us to develop state-exploration algorithms inside HOL like the reachability analysis, model checking and the invariant checking procedures. The transition and output relations are written as HOL terms. They are translated to their corresponding MDGs and then reachability analysis is performed. The state machines we consider are the abstract state machines which raises the level of abstraction of the problem specification. We also developed decision procedures based on the multiway decision graphs allowing the equivalence checking and tautology checking of a certain subset of HOL terms automatically. Finally we illustrated our approach by considering the Island Tunnel Controller example for which we verified a number of safety properties.

The embedding of the MDGs in HOL opens the way to the development of a wide range of new verification applications combining the advantages of state-exploration techniques and theorem proving. There are many opportunities for further work on this embedding and its use for formal verification. For instance, MDG canonicity can also be used in HOL for term simplification. In fact, when built, MDGs are reduced by construction. Retrieving the term represented by

this graph gives a simplification of the original term. The Embedding can be used for the formal proof of the soundness of the MDG algorithms. A similar work was done in [4] to verify a SPIN model checking algorithm. Finally, the embedding can be enhanced by using the LCF style. In this case, an MDG representation for a HOL term cannot be constructed by using the *termToMdg* function, instead, it is derived from inference rules, corresponding to MDG operators, and the trivial MDGs representing simple equations. This restricts the scope of soundness to single operators which are easy to get right [8].

References

1. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Theorem Proving in Higher-Order Logics*, volume 1690 of LNCS, pages 323–340. Springer-Verlag, 1999.
2. O. Ait Mohamed, X. Song, and E. Cerny. On the Non-termination of MDG-Based Abstract State Enumeration. *Theoretical Computer Science*, 300:161–179, August 2003.
3. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
4. C.-T. Chou and D. Peled. Verifying a Model-checking Algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of LNCS, pages 241–257. Springer-Verlag, 1996.
5. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
6. M. Gordon. Combining Deductive Theorem Proving with Symbolic State Enumeration. *21 Years of Hardware Formal Verification*, December 1998. Royal Society Workshop to mark 21 years of BCS FACS.
7. M. Gordon. Reachability Programming in HOL98 Using BDDs. In *Theorem Proving and Higher Order Logics*, volume 1869 of LNCS, pages 179–196. Springer-Verlag, 2000.
8. M. Gordon. Holbddlib Version 2, Documentation. Technical report, Computer Laboratory, Cambridge University, U.K., March 2002.
9. M. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
10. R. Harper. *Introduction to Standard ML*. School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1993.
11. J. Joyce and C. Seger. The HOL-Voss System: Model-Checking inside a General Purpose Theorem-Prover. In *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of LNCS, pages 185–198. Springer-Verlag, 1994.
12. C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4:123–193, 1999.
13. S. Kort, S. Tahar, and P. Curzon. Hierarchical Formal Verification Using a Hybrid Tool. *Software Tools for Technology Transfer*, 4(3):313–322, May 2003.
14. T. Kropf. *Introduction to Formal Hardware Verification*. Springer Verlag, 1999.
15. R. P. Kurshan. Formal Verification in a Commercial Setting. In *Proc. of Design Automation Conference*, pages 258–262, Anaheim, California, USA, June 1997.

16. M. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
17. R. Mizouni. Linking HOL Theorem Proving and MDG Model Checking. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2002.
18. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Automated Deduction*, volume 607 of LNCS, pages 748–752. Springer-Verlag, 1992.
19. V. Pisini. Integration of HOL and MDG for Hardware Verification. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2000.
20. S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model Checking with Automated Proof Checking. In *Computer Aided Verification*, volume 939 of LNCS, pages 84–97. Springer-Verlag, 1995.
21. K. Schneider and D. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-automata. In *Theorem Proving in Higher Order Logics*, volume 1690 of LNCS, pages 255–272. Springer-Verlag, 1999.
22. Y. Xu. *Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs*. PhD thesis, Computer Science Department, University of Montreal, Canada, 1999.

Formalizing the AMBA High Performance Bus

Malcolm C. Newey

The Australian National University, Canberra, ACT 0200, Australia
Malcolm.Newey@anu.edu.au

Abstract. This paper presents work in progress on a project to formalize the AMBA High Performance Bus (AHB) in higher order logic with a view to proving properties of the protocol, as a basis for verifying properties of computing components that might be connected to a bus and as a foundation for reasoning about *SoCs* (systems-on-a-chip).
This AMBA bus has been modeled using the specification language, Z [4, 5]. The system that is specified is one that consists of a number of masters and slaves connected by an AHB according to the protocol described in the document *"AMBA Specification (Rev. 2.0)"*[1].
The focus of this article is to present illustrative extracts from this Z specification[3] in order to exhibit a structure that arguably makes it a suitable foundation for the project as a whole.

Although much work has been done on the formalization of processors, the underlying assumption has usually been that they are directly connected to memory. In the world of the SoC (System-on-a-Chip), multiple components such as processors and memories on a chip communicate by means of a shared bus according to some protocol.

Traditionally, correct interactions on a bus are characterized in natural language with the help of timing diagrams. The aim of the present project is to describe the protocol using mathematics.

A commonly used industry standard is the Advanced High-performance Bus (AHB) flavour of the Advanced Microcontroller Bus Architecture (AMBA), produced by ARM. The standard reference for the AHB[1] is used as the principal authority for this document but the AMBA FAQ[2] was also consulted. We treat this FAQ as authoritatively correcting and clarifying the specification.

1 The AMBA High Performance Bus

Figure 1, which gives the standard communications view of a bus architecture, serves to illustrate that the active components of a typical system are classified as *masters* and *slaves*. Masters are such agents as CPUs and DMA devices; the typical slave is a memory, which may be on-chip or external.

In such a view, one master will acquire ownership of the bus while a transfer is completed over (at least)two clock cycles – one for adress and control signals to be sent and one for data. The AHB protocol overlaps these activities enabling the fast transmission of much larger blocks of data than can be sent in one atomic

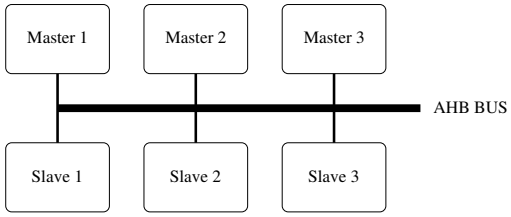


Fig. 1. Typical Bus Architecture User View

transfer. Such pipelining behaviour enables successive transfers to complete in consecutive clock cycles.

Figure 2, showing a typical trace of bus activity, shows the way that a master and slave interact to achieve this efficiency.

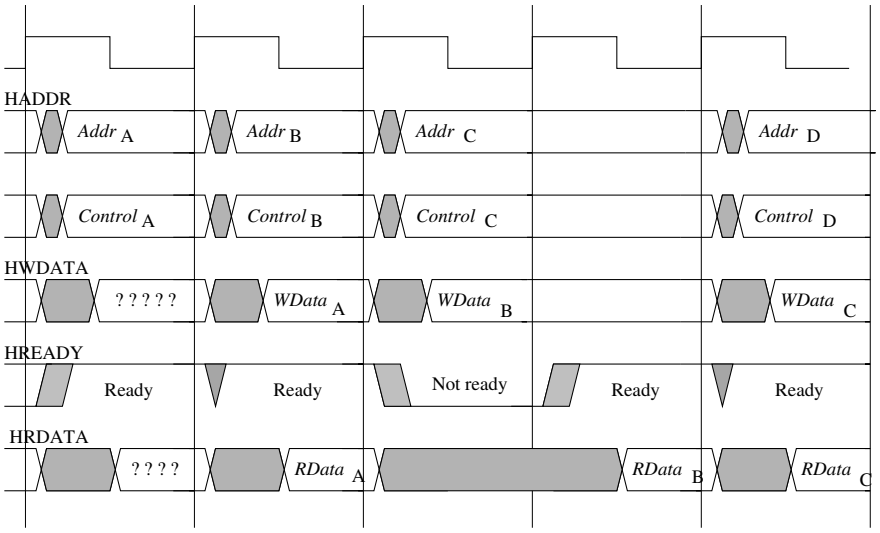


Fig. 2. Typical Timing Diagram Showing Data Transfer

Signals are carried on a significant number of separate bus lines; those shown in this timing diagram are the ones directly related to I/O and show three complete data transfers over five clock cycles. The first of these transfers consists of the signals with the subscript A, some of which are in the first clock cycle shown (the address phase) while the others are in the second cycle (the data phase).

Note that, for any given signal, the value that matters is the stable one at the end of any clock cycle; signals are sampled on each rising clock pulse.

– The signal value $Addr_A$ is the address of data for a transfer being initiated by the master that owns part of the bus including signal HADDR. This address is used to identify the slave whose responses will appear on another part of the bus in the next cycle.
– The signal $Control_A$ is a collection of more basic signals that describe the transfer here being initiated. For example, HSIZE gives the number of bits in parallel and HWRITE says whether a read or a write is intended.
– The signal HREADY originates in the slave that was addressed in the most recent transfer. In the case of the second cycle shown, it indicates that the slave addressed by $Addr_A$ has either consumed the signal $WData_A$ or produced the signal $RData_A$, as appropriate.
  In the case of the first clock period shown, the origin of the high HREADY signal is not apparent, but it is a necessary condition for us to interpret this cycle as the start of a transfer.
– The bus signal $RData_A$ originates in the slave addressed by $Addr_A$, not the slave that corresponds to $Addr_A$. Similarly, $WData_A$ flows from the master that owned the bus when the transfer commenced.

In the third clock cycle shown in Figure 2 the HREADY signal being low indicates that the slave addressed by $Addr_B$ is pleading for more time to complete the transfer. This prevents the master given the bus in cycle three from starting its intended transfer straight away. In the case that the transfer started in cycle two was a write, that master is obligated to hold the signal $WData_B$ for an extra cycle.

It is possible, if unlikely, that each of the transfers in Figure 2 was initiated by a different master. It should be clear that there is multiplexing of the signals from the various masters and slaves according to some scheme. The simplest, the multiplexing of address and control signals from a master while a transfer is getting under way, is shown in Figure 3.

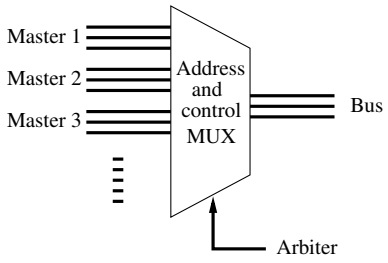


Fig. 3. Multiplexing Address and Control Signals from Masters to the Bus

The *Arbiter* controls the ownership of the bus allocating it to masters on the basis of requests, priorities and past history. The diagram shows how the Arbiter exerts control via the address/control multiplexer. The Arbiter also contributes

several signals to the bus, including one called HMASTER, which identifies, in any clock cycle, the master owning the bus .

An important concept in the design of the AHB is that of a burst, where a master gets control of the bus for the duration of a transaction where a larger block of memory than can be transmitted in one transfer is sent in consecutive transfers (typically 4, 8 or 16).

When a slave which is part way through a burst decides that it has more pressing business, it is able to pause the transaction and resume it when ready. This is referred to as a *split* and several signals on the bus relate to the splitting and resumption of bursts.

2 Modeling Basic Types

Since this paper relies on selected extracts from a full Z specification, it will be incomplete and so the reader should not expect that every item mentioned will be defined. In particular, the full specification[3] should be consulted for explanations of the various signals.

2.1 Masters and Slaves

Two sets of objects that are fundamental to the construction and operation of a system based around an AMBA bus are the sets of *masters* and *slaves,*. Of the 16 possible masters one is called *Dummy*. It is natural to model each of these sets as arbitrary subsets of $\{0..15\}$.

2.2 Basic Signal Values

All physical lines that are in a 'settled' state have values that are either *HIGH* or *LOW*. These values are synonyms for 1 and 0, respectively, in both our treatment and the specification from ARM.

$$Bit == \{ 0, 1 \}$$
$$HIGH == 1$$
$$LOW == 0$$

2.3 Numbers vs Bit Sequences

In this specification, many of the objects are coded as sequences of *Bits* with the convention that the last element of the sequence holds the least significant bit of a binary representation of a number. So, for example, the Z sequence $< 1, 0, 1, 1, 1 >$ will be interpreted as the number 23.

The function $bits2N$ maps bit-sequences to their numeric value.

$$bits2N : \text{seq } Bit \to \mathbb{N}$$

$$bits2N \langle\rangle = 0$$
$$\forall b : Bit;\ s : \text{seq } Bit \bullet bits2N(s \frown \langle b \rangle) = (2 * bits2N(s) + b)$$

2.4 The Decode Function

For any particular AMBA bus, the *Decoder* is a combinatorial circuit that selects a slave on the basis of the address lines. A crucial feature of the mapping is that every address in any 1024-byte block must map to the same slave.

$$
\begin{array}{|l}
decode : Address \rightarrow Slave \\
\hline
\forall\, a_1, a_2 : Address \bullet ((bits2N\ a_1) \operatorname{div} 1024) = ((bits2N\ a_2) \operatorname{div} 1024) \\
\qquad\qquad \Rightarrow\ decode(a_1) = decode(a_2) \\
\end{array}
$$

2.5 The Type of Data in Transfers

For any particular AHB bus, the width of each data path is a constant. Its value, *width*, must be one of the powers of two between 8 and 1024 bits, inclusive.

The size, in bits, of any transfer is bounded by *width* and must also be one of these numbers. It is convenient to have a global constant which is the set of permissible sizes for single transfers:

$$
\begin{array}{|l}
width : \mathbb{N} \\
permittedSizes : \mathbb{P}\,\mathbb{N} \\
\hline
width \in \{8, 16, 32, 64, 128, 256, 512, 1024\} \\
permittedSizes = \{n : \{8, 16, 32, 64, 128, 256, 512, 1024\} \mid n \leq width\} \\
\end{array}
$$

2.6 Signals and Signal Groups

The bus is composed of 18 groups of lines (apart from the clock), where each line carries one bit of information. Their names are the following (except for HLOCK and HBUSREQ).

$$
\begin{aligned}
Name ::= {} & HRESETn \mid HADDR \mid HTRANS \mid HWRITE \\
\mid {} & HSIZE \mid HBURST \mid HPROT \mid HWDATA \\
\mid {} & HSELx \mid HRDATA \mid HREADY \mid HRESP \\
\mid {} & HBUSREQx \mid HBUSREQ \mid HLOCKx \mid HLOCK \\
\mid {} & HGRANTx \mid HMASTER \mid HMASTLOCK \mid HSPLITx
\end{aligned}
$$

The names of signals originating from each master are as follows. The signals HBUSREQ and HLOCK from each master contribute one bit to the bus signals HBUSREQx and HLOCKx.

$$
\begin{aligned}
MSNames == \{ & HADDR, HWDATA, HTRANS, HWRITE, \\
& HSIZE, HBURST, HPROT, HBUSREQ, HLOCK \}
\end{aligned}
$$

Similarly, each slave has signals that carry data that is particular to that slave. They are multiplexed onto the bus when appropriate.

$$
SSNames == \{ HRDATA, HREADY, HRESP, HSPLITx \}
$$

One fundamental characteristic of each of these named groups of lines is the number of bits of the bus, a master or a slave that it occupies. The function $grpWidth$ gives the number of bits associated with any given signal name.

$$
\begin{array}{|l}
grpWidth : Name \to \mathbb{N} \\
\hline
grpWidth \;=\; \{\; (HRESETn \mapsto 1),\;\; (HADDR \mapsto 32),\;\; (HTRANS \mapsto 2), \\
\qquad (HWRITE \mapsto 1),\;\; (HSIZE \mapsto 3),\;\; (HBURST \mapsto 3), \\
\qquad (HPROT \mapsto 4),\;\; (HWDATA \mapsto width),\;\; (HSELx \mapsto \#Slave), \\
\qquad (HRDATA \mapsto width),\;\; (HREADY \mapsto 1),\;\; (HRESP \mapsto 2), \\
\qquad (HBUSREQx \mapsto \#Master),\;\; (HBUSREQ \mapsto 1), \\
\qquad (HLOCKx \mapsto \#Master),\;\; (HLOCK \mapsto 1), \\
\qquad (HGRANTx \mapsto \#Master),\;\; (HMASTER \mapsto 4), \\
\qquad (HMASTLOCK \mapsto 1),\;\; (HSPLITx \mapsto 16)\}
\end{array}
$$

2.7 Signal Maps

Each signal is a bunch of bits and so we model each of them as an object of type *'sequence of bits'*. Consequently, the state of the bus, each master and each slave is modeled as a mapping from names to such bit sequences.

$$
\begin{array}{|l}
SignalMap : \mathbb{P}(Name \pfun \operatorname{seq} Bit) \\
\hline
\forall\, s : SignalMap \;\bullet\; \forall\, n : \operatorname{dom} s \;\bullet\; \#(s\;\; n) = grpWidth(n)
\end{array}
$$

Each of the types $MapM$, $MapS$ and Bus is a specialization of $SignalMap$ as appropriate for masters, slaves and the bus itself.

Data Width The signal HSIZE indicates the number of lines of the data bus that will be used in the current transfer. The 3-bit value of HSIZE can be decoded using the following function:

$$
\begin{array}{|l}
decodeSize : (0\,..\,7) \to \mathbb{N} \\
\hline
decodeSize = \{(0 \mapsto 8), (1 \mapsto 16), (2 \mapsto 32), (3 \mapsto 64), \\
\qquad\qquad (4 \mapsto 128), (5 \mapsto 256), (6 \mapsto 512), (7 \mapsto 1024)\}
\end{array}
$$

3 State-Related Types

3.1 The Cycle Abstraction

In the following schema, the state of all signals at any instant is captured. The structure of the abstraction reflects the separation of signals between the various parts of the system. The predicate part expresses the following properties:

1. Arbitration results in exactly one master being granted the bus;
2. Exactly one bit of HSELx will be high;
3. Address and control signals for current master are multiplexed on the bus;

4. Data buses and the HREADY and HRESP signals are multiplexed;
5. The HSPLIT signals are properly multiplexed;
6. Certain signals from the dummy master can be relied on.

Cycle

$$
\begin{array}{l}
bus : Bus \\
stateMx : Master \rightarrow MapM \\
stateSx : Slave \rightarrow MapS \\
\hline
\exists_1\, m : Master \bullet (bus\ HGRANTx)(16 - m) = HIGH \\
\forall\, s : Slave \bullet (bus\ HSELx)(16 - s) = HIGH \\
\qquad \Leftrightarrow (bus(HADDR) \mapsto s) \in decode \\
\textbf{let}\ map == stateMx(bits2N(bus\ HMASTER)) \bullet \\
\quad ((\forall\, nam : \{HADDR, HTRANS, HWRITE, HSIZE, \\
\qquad HBURST, HPROT\} \bullet (bus\ nam) = (map\ nam)) \land \\
\quad (\textbf{let}\ mast == bits2N(bus(HMASTER)) \bullet \\
\qquad ((bus\ HLOCKx)(16 - mast) = (map\ HLOCK)(1) \land \\
\qquad (bus\ HBUSREQx)(16 - mast) = (map\ HBUSREQ)(1)))) \\
\exists\, m : Master \bullet bus(HWDATA) = stateMx(m)(HWDATA) \\
\forall\, nam : \{HRDATA, HREADY, HRESP\} \bullet \\
\quad \exists\, s : Slave \bullet bus(nam) = stateSx(s)(nam) \\
\forall\, m : Master \bullet (((bus\ HSPLITx)(16 - m) = HIGH) \\
\qquad \Leftrightarrow (\exists\, s : Slave \bullet stateSx(s)(HSPLITx)(16 - m) = HIGH)) \\
(bus\ HBUSREQx)(16 - Dummy) = LOW \\
(bus\ HLOCKx)(16 - Dummy) = LOW \\
stateMx(Dummy)(HTRANS) = IDLE
\end{array}
$$

For each component of the system, history is important for constraining future behaviour. However, that dependence does not involve the relative timing of the possibly numerous events within the current or past clock cycles. So when the Cycle abstraction is used as a unit of history it will a snapshot of all signals at the rising clock pulse.

3.2 The Transfer Abstraction

Transfers consist of two or more consecutive cycles. The first cycle is the one where a master owns the bus and completes the address phase of the transfer. The last cycle is where the transfer of data is complete. All the intermediate cycles in the transfer are wait cycles that arise because the slave delays by forcing HREADY low.

The predicate part of the schema for the type *Transfer*, which follows, asserts consistency of all cycles making up the transfer, not just the first and last.

Transfer

$cycles : \text{seq}\ Cycle$
$time : \mathbb{N}$

$\#cycles > 1$
let $trans == ((cycles\ 1).bus)(HTRANS);$
$addr == bits2N(((cycles\ 1).bus)(HADDR));$
$size == decodeSize(bits2N(((cycles\ 1).bus)(HSIZE)))\ \bullet$
$(trans \in \{\ NONSEQ, SEQ\ \}$
$\land\ addr \bmod (size \operatorname{div} 8)\ =\ 0$
$\land\ addr \operatorname{div} 1024\ =\ (addr + (size \operatorname{div} 8) - 1) \operatorname{div} 1024$
$\land\ (\forall j : (2\,..\,\#cycles)\ \bullet\ ((cycles\ j).bus)(HRESP) = OKAY)$
$\land\ ((last\ cycles).bus)(HREADY) = \langle HIGH \rangle$
$\land\ (\forall j : (2\,..\,(\#cycles - 1))\ \bullet$
$((cycles\ j).bus)(HREADY) = \langle LOW \rangle)$
$\land\ (((cycles\ 1).bus)(HWRITE)\ =\ \langle HIGH \rangle)\ \Rightarrow$
$(\forall j : (2\,..\,\#cycles)\ \bullet$
$((cycles(j)).bus)(HWDATA) = ((cycles\ 2).bus)(HWDATA)))$

3.3 AMBA Transactions

A complete transaction consists either of a single transfer or of some number of transfers which accomplish the transmission of one block of data in uniform sized pieces. Details of this breakup of a transaction are given by the HBURST signal. The first transfer will be tagged NONSEQ and subsequent ones will have SEQ as the HTRANS signal.

The components of any *Transaction* object are the sequence of transfers that belong to it with a flag to indicate completion (or otherwise). The possible values for this flag are *InProgress*, *Complete*, *Split* and *Interrupted*.

Transaction

$xfers : \text{seq}\ Transfer$
$completion : Completion$

$\#xfers > 0$
let $mode == modeOf(xfers\ 1)\ \bullet$
$(\#xfers = beatUB(mode)\ \Rightarrow\ completion = Complete\ \land$
$\#xfers < beatUB(mode)\ \Rightarrow\ completion \neq Complete\ \land$
$(mode = INCR)\ \Rightarrow\ completion \neq Interrupted\ \land$
$addressesOf(xfers) \subseteq$
$addrSeq(addrOf(xfers(1)), sizeOf(xfers(1)), mode))$
$\forall t : Transfer\ \mid\ t\ \in\ (\text{ran}\ xfers)\ \bullet\ xMatches(t, (xfers\ 1))$

Functions appearing without definition in this schema are described thus:

- $modeOf(t)$ extracts the burst mode of transfer t (its HBURST value).

- $beatUB(m)$ is a bound on the length of a burst of mode m.
- $addrSeq(A, s, m)$ gives the full sequence of transfer addresses starting with A in a burst of mode m and transfer width size s bits.
- Two transfers are in relation *xMatches* if their control signals match.

4 State Specification Schemas

The state of an AMBA bus is characterized by what progress has been made toward finalizing the current cycle. The physical aspects of state are the values of signals on the bus, on the masters and on the slaves, but the behaviour of the bus depends also on the past history of the system. The latter aspects are modeled in the schema *Histories* defined below.

4.1 The History Component

The following schema captures the history of the system at the levels of cycles, transfers and transactions. Note that there is redundancy in that transactions are all made up of transfers which are made up of cycles.

Histories

$$
\begin{array}{l}
cycHist : \text{seq } Cycle \\
xferHist : \text{seq } Transfer \\
xactionHist : \text{seq } Transaction \\
partXfer : \text{seq } Cycle \\
partXactions : Master \pfun Transaction \\
\hline
\exists\, sc : \text{seq } Cycle \bullet (sc \frown partXfer) = cycHist \\
\exists\, t : Transfer \bullet partXfer = front(t.cycles) \\
\forall\, j, k : \text{dom } xferHist \bullet j < k \Rightarrow \\
\qquad ((xferHist\ j).time) < ((xferHist\ k).time) \\
\forall\, t : Transfer \bullet t \in (\text{ran } xferHist) \Leftrightarrow \\
\qquad (\forall\, j : (\ 1 \,..\, \#(t.cycles)\ ) \bullet cycHist(t.time + j - 1) = (t.cycles)(j)) \\
\forall\, t : (\text{ran } partXactions) \bullet \\
\qquad t.xfers \neq \langle\rangle \ \land\ t.completion \in \{\ InProgress, Split\ \} \\
\forall\, m : \text{dom } partXactions \bullet masterOf(((partXactions\ m).xfers)(1)) = m \\
\#(\{t : \text{ran } partXactions \mid t.completion = InProgress\}) \leq 1 \\
\forall\, t : \text{ran } xactionHist \bullet t.completion \in \{Complete, Interrupted\} \\
\text{ran } xactionHist \cap \text{ran } partXactions = \varnothing \\
\text{ran } xactionHist = \text{ran } xactionHist \cup \text{ran } partXactions
\end{array}
$$

In the above schema the predicates assert that

- The partial transfer, if any, is the tail of the cycle history.
- An appropriate next cycle can legally complete *partXfer*.
- The transfer history, *xferHist*, is properly ordered.

– *xferHist* is complete and consistent, relative to the cycle history.
– *partXactions* is well formed.
– Each transaction in *xactionHist* is either Complete or Interrupted.
– Each transfer in *xferHist* is in either in *xactionHist* or in a *partXaction*.

4.2 The State in General

The schema *AMBA_State* captures system state to extent of its current snapshot and a complete record of past cycle activity. Since this state schema contains both aspects, its predicate part adds constraints on possible values of current signals to that given in the *Cycle* abstraction. In particular,

– The HWDATA signal is multiplexed from the master that owns the data bus at that point, not necessarily the current master.
– Several signals that originate in slaves are multiplexed from the slave that was addressed in the last transfer.

$$
\begin{array}{|l}
\hline
\textit{AMBA\_State} \\
\textit{Cycle} \\
\textit{Histories} \\
\hline
\textit{partXfer} \neq \langle\rangle \ \Rightarrow \\
\quad (\textbf{let}\ \textit{lastMast} == \textit{bits2N}(((\textit{partXfer}\ 1).\textit{bus})(\textit{HMASTER})); \\
\qquad \textit{lastSlav} == \textit{decode}(((\textit{partXfer}\ 1).\textit{bus})(\textit{HADDR}))\ \bullet \\
\qquad (\textit{bus}(\textit{HWDATA}) = \textit{stateMx}(\textit{lastMast})(\textit{HWDATA})\ \wedge \\
\qquad\ \textit{bus}(\textit{HRDATA}) = \textit{stateSx}(\textit{lastSlav})(\textit{HRDATA})\ \wedge \\
\qquad\ \textit{bus}(\textit{HREADY}) = \textit{stateSx}(\textit{lastSlav})(\textit{HREADY})\ \wedge \\
\qquad\ \textit{bus}(\textit{HRESP}) = \textit{stateSx}(\textit{lastSlav})(\textit{HRESP})\ )) \\
\hline
\end{array}
$$

5 Intracycle Operations

Most atomic events that occur in the system are actions initiated by masters and slaves updating one or more of the signals for which they are the source. The more unusual event is initiated asynchronously by the reset controller. An action by any one of these agents will cause changes to the state of the bus as specified in the *BasicOperation* schema given subsequently.

5.1 The Operation Abstraction

Each primitive operation is characterized by identifying the agent (a master, a slave or the reset controller) and a map which indicates the updates to that agent's signals. Primitive operations are identified by their structure as given in the type *OpType* below.

The possible values that the agent identifier variable (*agId*) can take depends on the sort of that agent. These agent types are distinguished with the type

$$AgentType ::= Mast \mid Slav \mid Bus$$

OpType

$$agTy : AgentType$$
$$agId : \mathbb{N}$$
$$updates : SignalMap$$

$$agTy = Mast \Rightarrow (agId \in Master) \land (\exists\, m : MapM \bullet updates \subseteq m)$$
$$agTy = Slav \Rightarrow (agId \in Slave) \land (\exists\, m : MapS \bullet updates \subseteq m)$$
$$agTy = Bus \Rightarrow agId = 0 \land updates = \{ HRESETn \mapsto \langle LOW \rangle \}$$

5.2 Application of Basic Operations

Each basic operation, other than a reset, is effected by a master or a slave altering its own signal map while leaving those of all others the same. Multiplexing and decoding cause signals from some agents to make it onto the bus thus:

- The Cycle schema (imported through Amba_State) takes care of the the relationship between old and new multiplexed address and control signals (HADDR, HTRANS, HWRITE, HSIZE, HBURST and HPROT).
  It also captures the derivability of HSELx, HSPLITx, HBUSREQx and HLOCKx from a variety of other signals.
- The invariant of schema AMBA_State specifies how multiplexing of signals HWDATA, HRDATA, HREADY and HRESP happens.
- The remaining bus signals are specified directly.

BasicOperation

$$\Delta AMBA\_State$$
$$\Xi Histories$$
$$op? : OpType$$

$$op?.agTy = Mast \Rightarrow$$
$$(stateSx' = stateSx \land bus'(HRESETn) = bus(HRESETn) \land$$
$$stateMx' = stateMx \oplus$$
$$\{ op?.agId \mapsto (stateMx(op?.agId) \oplus op?.updates) \})$$
$$op?.agTy = Slav \Rightarrow$$
$$(stateMx' = stateMx \land bus'(HRESETn) = bus(HRESETn) \land$$
$$stateSx' = stateSx \oplus$$
$$\{ op?.agId \mapsto (stateSx(op?.agId) \oplus op?.updates) \})$$
$$op?.agTy = Bus \Rightarrow (bus'(HRESETn) = \langle LOW \rangle \land$$
$$stateMx' = stateMx \land stateSx' = stateSx)$$
$$\forall\, nam : \{ HGRANTx,\ HMASTER,\ HMASTLOCK \} \bullet$$
$$bus'(nam) = bus(nam)$$

6 End of Cycle Operations

The end of each clock cycle is marked by the rising edge of a clock pulse. All signals will be steady at that time and for some minimum hold time after the rising edge. Although no operations that change signals should take place at that time, our interpretation of the state changes because we deem transfers to start and/or complete at cycle's end. Thus we define an operation *HistoryUpdate* which registers in the *Histories* component of state just what has cumulatively been accomplished in the system.

There are three matters for *HistoryUpdate* to address:

- Transfers that may have completed or been extended;
- Transactions that may be more advanced or may be completed, interrupted or split.
- Transfers that may have just commenced with their control/address phase.

It makes sense to separate these concerns into three sub-operations which are carried out sequentially and so we define *HistoryUpdate* in terms of three new schemas which will be defined in subsequent sub-sections:

$$HistoryUpdate \;==\; ActiveTransferUpdate \fatsemi XactionActivity \fatsemi NewTransfer$$

This sequential decomposition requires a minimum of information discovered in one phase to be transmitted to later phases (apart from that naturally contained in the variables of global state). The variable *xferInProgress* is an output of *ActiveTransferUpdate* and an input to *XactionActivity*.

6.1 Transfer Continuation and Completion

If there was an incomplete transfer cycle after the last clock pulse then this transfer may be complete, incomplete are may have been aborted. So there are four cases for *ActiveTransferUpdate* to consider; the four cases, handled by four sub-operation schemas, and combined thus:

$$\begin{aligned} ActiveTransferUpdate \;==\; & TransferContinuation \lor TransferCompletion \lor \\ & TransferAborted \lor NoCurrentTransfer \end{aligned}$$

Without considering violations of the AMBA protocol, the following table gives the precise preconditions for the various cases.

Partial transfer	HREADY	HRESP	Relevant Schema
No	Don't care	Don't care	*NoCurrentTransfer*
Yes	low	Don't care	*TransferContinuation*
Yes	high	OKAY	*TransferCompletion*
Yes	high	not OKAY	*TransferAborted*

Each of the schemas given in the classify the partial transfer, if any, and give an output value which is of type *XferStatus* which is as follows:

$$XferStatus ::= None \mid Extending \mid Finished \mid Aborted$$

A Representative Example - *TransferContinuation*

At the end of a cycle a transfer is deemed to be continuing if the slave involved in the transfer is pleading not ready. This usually indicates an extension of the transfer by the slave but it also includes the situation where the slave is indicating first cycle of a two-cycle READY, ERROR or SPLIT response.

TransferContinuation

$\Delta AMBA\_State$
$\Xi Cycle$
$xferInProgress! : XferStatus$

$partXfer \neq \langle\rangle$
$bus(HREADY) = \langle LOW \rangle$
$\#partXfer > 1 \ \wedge \ bus(HRESP) \neq OKAY \ \Rightarrow$
$\quad ((last\ partXfer).bus)(HRESP) = OKAY$
$cycHist' = cycHist \frown \langle \theta Cycle \rangle$
$xferHist' = xferHist$
$partXfer' = partXfer \frown \langle \theta Cycle \rangle$
$xactionHist' = xactionHist \ \wedge \ partXactions' = partXactions$
$xferInProgress! = Extending$

6.2 Transaction Continuation and Completion

When it comes to updating history to reflect progress (or otherwise) in transaction activity there are three broad possibilities – a transaction may be completed (successfully or not), a transfer may grow larger but still be partial, or there may be no change to any transaction. In writing the schemas however, it is convenient to specify *XactionActivity* in terms of ten separate cases, as follows.

In the first table each row starts with the value of the variable *xferInProgress*. The columns labeled HTRANS and HBURST give the values of those signals in the partial transfer, if any, rather than in the current state,

Xfer Status	HTRANS	HBURST	HRESP	Relevant Schema
Extending	Don't care	Don't care	OKAY	*MidTransfer*
None	N/A	N/A	OKAY	*BusIdle*
Finished	NONSEQ	SINGLE	OKAY	*SingleShot*
Finished	NONSEQ	not SINGLE	OKAY	*FirstOfBurst*
Aborted	SEQ	not SINGLE	SPLIT	*SplitBurst*
Aborted	Don't care	Don't care	RETRY	*InterruptedBurst*
Aborted	Don't care	Don't care	ERROR	*InterruptedBurst*

The following table distinguishes between the various remaining cases; in each of them the variable *partXfer* contains a completed transfer . The first row corresponds to the case that the burst is continuing while other rows are for the several ways in which that completed transfer could be the last in a burst.

HTRANS	Last type	This type	New master	Relevant Schema
SEQ, BUSY	not SINGLE	not SINGLE	No	*BurstContinues*
Don't care	INCR	Don't care	Yes	*IncrBurstDone*
NONSEQ, IDLE	INCR	Don't care	No	*IncrBurstDone*
NONSEQ, IDLE	not INCR	Don't care	Don't care	*BurstComplete*
Don't care	not INCR	Don't care	Yes	*PreemptedBurst*

An Example - The schema *MidTransfer*

In this case, the variable *xferInProgress* would have been set to *Extending* by the previous history update operation *TransferContinuation* and so there is nothing to do.

MidTransfer

$$\Xi AMBA\_State$$
$$xferInProgress? : XferStatus$$

$$xferInProgress? = Extending$$

A Bigger Example - The Schema *SingleShot*

In the case that a transfer just completed was of burst mode SINGLE, this constitutes a complete transaction consisting of that one transfer. It is not possible that there is a partial transfer for the same master, since it would have been completed or aborted on a previous cycle.

SingleShot

$$\Delta AMBA\_State$$
$$\Xi Cycle$$
$$xferInProgress? : XferStatus$$
$$m : Master$$
$$new : Transaction$$

$$xferInProgress? = Finished \ \land \ bus(HRESP) = OKAY$$
$$m = bits2N(((partXfer\ 1).bus)(HMASTER))$$
$$m \notin \mathrm{dom}\, partXactions$$
$$((partXfer\ 1).bus)(HTRANS) = NONSEQ$$
$$((partXfer\ 1).bus)(HBURST) = SINGLE$$
$$new.xfers = \langle partXfer \rangle$$
$$new.completion = Complete$$
$$cycHist' = cycHist \ \land \ xferHist' = xferHist$$
$$partXfer' = \langle\rangle$$
$$xactionHist' = xactionHist \frown \langle new \rangle$$
$$partXactions' = partXactions$$

6.3 Transfer Initiation

The schema *NewTransfer* updates history according to whether the control signals indicate that a new transfer was started in the cycle just completed (or not). Each of these two possibilities is captured by a schema and so *NewTransfer* is defined thus:

$$NewTransfer \;==\; TransferInitiation \lor NoNewTransfer$$

The following table shows necessary conditions for each of the following two schemas to be applicable.

HREADY	HTRANS	*partXfer*	Relevant Schema
LOW	Don't care	Don't care	*NoNewTransfer*
HIGH	SEQ, NONSEQ	empty	*TransferInitiation*
HIGH	SEQ, NONSEQ	nonempty	*NoNewTransfer*
HIGH	BUSY, IDLE	empty	*NoNewTransfer*
HIGH	BUSY, IDLE	nonempty	Impossible

When a new transfer is initiated, the signal HTRANS having value SEQ occurs exactly when the new transfer will augment an existing partial transaction for the current master. In such a case the control and address signals must be consistent with this partial transaction.

TransferInitiation

$\Delta AMBA\_State$
$\Xi Cycle$
$m : Master$
$trans : \{\ SEQ, NONSEQ\ \}$
$new : Transfer$

$trans = bus(HTRANS)$
$m = bus(HMASTER)$
$partXfer = \langle\rangle$
$bus(HREADY) = HIGH$
$trans = SEQ \;\Leftrightarrow\; m \in (\mathrm{dom}\, partXactions)$
$partXfer' = \langle \theta Cycle \rangle$
$(new.cycles) = partXfer'$
$trans = SEQ \;\Rightarrow xMatches(new, (partXactions\ m)(1))$
$cycHist' = cycHist$
$xferHist' = xferHist$
$xactionHist' = xactionHist$
$partXactions' = partXactions$

7 The Composite Cycle Operation

Given above are schemas for all aspects of what can happen within one cycle. The following schema, *CompleteCycle* specifies what happens, over a full clock cycle when a sequence of basic operations occurs.

$$CompleteCyle \;==\; Arbitration \fatsemi OperationSequence \fatsemi HistoryUpdate \\ \lor \quad ResetCycle$$

Space limitations in the present paper preclude a discussion of the operation of the arbiter or what happens in a clock cycle where a reset occurs.

8 Acknowledgements

The author is grateful for the support of the Computer Lab at Cambridge University where this work was started and wishes to thank Mike Gordon for discussions that gave rise to this work.

9 Conclusion

Although the approach of modeling an AMBA bus using Z was taken to get the project of the ground quickly, it has proved to be a fortunate decision. The obvious alternative was to formalize AHB directly in higher order logic using HOL since theorem proving in that system was definitely on the agenda. There are two reasons for satisfaction with postponing HOL activity.

- The AMBA buses turn out to be quite complicated to understand in detail. The Z spec. is 30 pages of which fully half is mathematics. Because of the complexity, the development process saw the document go through seven versions as various ways of looking at the problem were tried. Had the same development been done in HOL (or any similar system) lots of time would surely have been wasted proving theorems associated with the dead ends.
- Having a formal specification in Z makes the audience of possible readers much wider than if it was presented as a proof script for a theorem prover. Of course, having parallel definitions in two formal systems begs the question of verifying consistency. However, this question has been asked before and so we need to search for possible answers for the present situation.

References

1. ARM Limited: *AMBA Specification (Rev. 2.0).* ARM Limited, IHI-0011A, May 1999. `http://www.arm.com/armtech/AMBA_Spec?OpenDocument.`
2. ARM Limited: *AMBA FAQ.* ARM Limited, Last updated 23 Jan. 2001. `http://www.arm.com/support/amba?OpenDocument.`
3. M. C. Newey: *A Z Specification of the AMBA High Performance Bus.* Draft version, June 2004. `http://cs.anu.edu.au/Ñalcolm.Newey/AMBA/AHB.v7.pdf`
4. J. Woodcock and J. Davies: *Using Z.* (ISBN 0-13-948472-8). Prentice Hall International Series, 1996.
   Also `http://www.usingz.com/.`
5. J.M. Spivey: *The Z Notation - A Reference Manual.* Prentice Hall International Series, 2nd edition, 1992.
   Also `http://spivey.oriel.ox.ac.uk/~mike/zrm/.`

Implementing the Calculus of Inductive Constructions in the MetaPRL framework

Natalia Novak and Yegor Bryukhov

Graduate Center, City University of New York
365 Fifth Avenue, New York, NY 10016
nnovak@gc.cuny.edu
ybryukhov@gc.cuny.edu

Abstract. The Calculus of Inductive Constructions is an underlying logic of the Coq proof assistant - a widely used mature proof assistant. In this paper we present our work on implementing the Calculus of Inductive Constructions in the MetaPRL logical framework. Rules from the Coq reference manual have quite unrestricted format so we have to make certain design decisions in order to express those rules in the plain Gentzen style supported by MetaPRL. The most complicated case-analysis and fixpoint rules have yet to be implemented. There is a working implementation with rudimentary proof automation; the toy example of inductive definition (parameterized lists) is type-checked.

1 Introduction

MetaPRL [5,7] is a relatively young logical framework from the PRL family [2] originated at Cornell University.

Among logical theories already defined in MetaPRL there are

- NuPRL-like Computational Type Theory CTT (based on Martin-Löf's Intuitionistic Type Theory);
- the constructive set theory CZF, based on Aczel's axiomatization;
- the First Order Logic.

MetaPRL was designed to address scalability and efficiency issues of NuPRL; as a result of these efforts CTT in MetaPRL is two decimal orders of magnitude faster than in NuPRL [6].

The Coq proof assistant [8] is a widely used mature logical system. Its underlying logic is the Calculus of Inductive Constructions (CIC) [8,3,4,10,1]. CIC is a rather sophisticated and powerful system. Implementing CIC in MetaPRL is the natural next step in developing the latter. It would be a good test for MetaPRL's universality and a challenge for a fast MetaPRL proof engine. It could help MetaPRL to import Coq's vast formal libraries.

In this paper we discuss our pilot implementation of CIC in the MetaPRL logical framework. We have a working code (rules, rewrites and tactics) that implements lambda calculus and inductive definitions. Implementation of inductive

definitions is not complete. We implemented rule about correctness of an inductive definition, typechecking of inductive types and constructors. Case-analysis and fixpoint are not supported yet.

2 MetaPRL meta-language

A brief syntax description of MetaPRL will give a better understanding of implementation problems and their solutions further in the article. *Terms* have the following syntax:

$$term ::= operator\ \{\ bterms\ \}$$

where the *operator* represents the *name* of a term and *bterms* are possibly bound terms.

Bound terms have the following syntax:

$$bterm ::= term \mid vars.term$$

For bound term $v_1, \ldots, v_n.t$ variables $v_1, \ldots, v_n$ are bound in t. Such binding is the part of signature (arity) of the outer operator. For example, $\forall x : T.P(x)$ can be expressed as $\mathsf{forall}\{T; x.P[x]\}$, where forall has arity (0,1) - no bindings in the first subterm and one binding in the second subterm.

Variables are special terms treated specifically by the system. There are two types of variables: *first-order* variables represent variables of the *object theory*, *second-order* variables (*meta-level* variables) represent terms with substitutions.

A theory is defined by its inference rules and computational equivalences between terms. The syntax of an inference rule is

$$\textbf{rule}\ name\ [params] : inference$$

where *name* is the name of the rule, *params* are extra parameters passed to the inference rule (optional) and *inference* is a valid inference in the defined logic. Inference is declared in the following form:

$$inference ::= term \mid term \rightarrow inference$$

Inference rules can be derived from previous rules or they can be defined as a primitive axioms of the theory.

Rewrites can be used to establish computational and/or definitial equality between certain terms. Rewrites are declared as follows:

$$\textbf{rewrite}\ name\ [params] : [conditions]\ redex \leftrightarrow contractum$$

where *name* is a name, *params* are extra variables and terms needed in rewrite and if the rewrite is conditional then the condition is stated in *conditions*. Rewrite replaces redex with contractum in any context. Just like rules rewrites can be primitive or derived. Rewrites and inference rules are logical inferences of MetaPRL.

Sequent schema language [9] is used for specifying new inference rules in a theory. The extension of the theory with sequents is conservative and derived rules can be used as primitive axioms [9]. The *sequent* syntax is:

$$\textbf{sequent}[name]\{H_1; \ldots; H_n \vdash C\}$$

where *name* is a name of a sequent (optional), which can be used to assign different semantics to differently named sequents. Each of $H_1; \ldots; H_n$ is either a variable declaration (hypothesis) or a sequent context, and C is a conclusion. Contexts are meta-variables that are used as placeholders for sequences of hypotheses (again variable declarations and contexts). A variable declaration $x : T$ introduces a variable x bound in the rest of the sequent.

One can think of sequents as a special kind of terms with flexible arity, where *name* is an analogue of operator and "sequent" indicates that this is a special kind of term (with flexible arity). It is more convenient to look at sequents in this perspective for the rest of the article.

There is a discipline of specifying permitted dependencies of a context or a second-order variable on all contexts and declarations from the left of it. We say that a context Γ (a second order variable A) can depend on variable declaration $x : T$ if x is allowed to occur in Γ (in A). We indicate it by $\Gamma[x]$ and $A[x]$. If variable declared before Γ (before A) is not listed in brackets it is interpreted as prohibition of free occur

We say that a context Δ (a second order variable A) can depend on a context Γ if it is allowed for variables potentially declared in Γ to occur freely in Δ (in A). We indicate it by $\Delta_{\{\Gamma\}}$ and $T_{\{\Gamma\}}$. If a context Γ declared before Δ (before A) and Γ is not listed in curly brackets after Δ (after A) it is interpreted as prohibition of occur potentially declared in Γ to occur freely in Δ (in A). If curly brackets are not used at all it is interpreted as a dependency on *all* preceding contexts.

Sequents are legitimate terms and can be used wherever regular terms can be used. In particular nested sequents (when conclusion is again a sequent) allows to separate different kinds of contexts from each other so they won't mix:

$$\textbf{sequent}\{\Gamma_A \vdash \textbf{sequent}\{\Gamma_B \vdash C\}\}$$

can be thought as $\Gamma_A|\Gamma_B \vdash C$ where "|" is a marker used to enforce some structure in antecedent pattern (to separate Γ_A and Γ_B).

3 A brief description of CIC

CIC is based on a typed lambda calculus. Without inductive definitions it's a system $\lambda P\omega$ (or λC) from Barendergt's cube. There is no syntactical differentiation between types and objects, they are just terms. Terms are built from variables, global names, constructors, abstraction, application, product and "let-in" expressions. Each term should have a type, types of types are constants called sorts. There are two basic sorts Set and Prop and a cumulative hierarchy of

higher sorts $\mathsf{Type(0)}, \mathsf{Type(1)}, \ldots$ all containing the basic sorts. Intuitively Prop is a type of all propositions and Set is a type of specifications (of programs) and usual types (integers, booleans, lists, etc).

We based our work on the system of rules presented in the chapter 4 "Calculus of Inductive Constructions" of The Coq Proof Assistant Reference Manual [8]. Although CIC is formulated in Gentzen style, it's not a usual plain Gentzen style system. Each CIC rule is explicitly parameterized with environment and can explicitly change it. Environment contains declarations of global constants and global assumptions. Such a non standard format is chosen because of inductive definitions - once inductive definition is verified to be correct, all types and constructors it defines are (automatically) added to the environment. Alternatively one can carry the whole inductive definition all over the proof as a term. The latter approach is in original papers [4,10] about inductive definitions for the Calculus of Constructions; it is (at least) easier to express in the plain Gentzen style. For this reason we use the latter approach.

4 Implementation problems and their solutions

Coq's implementation of CIC operates with the notion of *environment* (or to be more precise *global environment*). It is an ordered list of declarations of global names, such as names of new operators and types. Of course MetaPRL maintains something similar internally but it is not available for the direct control of the user. It also seems that explicit global environment was introduced primarily for efficiency reason - to mention inductive definitions only once and later only refer to them. We prefer the global environment. So we modified all rules not to use the global environment explicitly.

There is also a notion of *context* (or more precisely *local context*) where the names of variables are declared. Contexts are native entities of the MetaPRL meta-language so we are fine here.

There are two official kinds of judgments in CIC:

- $E[\Gamma] \vdash t : T$ means that the term t is well-typed and has type T in the environment E and context Γ
- $\mathcal{WF}(E)[\Gamma]$ means that the environment E is well-formed and the context Γ is a valid context in this environment

But in the actual rules we find one more kind of judgement:

- $D \in E$

where D is either inductive definition $\mathsf{Ind}(\Gamma)[\Gamma_P]\{\Gamma_I := \Gamma_C\}$ or constant declaration $c : T$ and E is an environment. It means that E is well-formed and contains D (or if $D \in E$ is the conclusion of the rule, D is added to E).

4.1 $\mathcal{WF}$-judgement

More traditional formulation of calculus of constructions [1] does not use $\mathcal{WF}$-judgement:

$$\vdash \mathsf{Prop} : \mathsf{Type(i)} \qquad \vdash \mathsf{Set} : \mathsf{Type(i)} \qquad \vdash \mathsf{Type(i)} : \mathsf{Type(j)} \qquad \text{axioms, } i < j$$

$$\frac{\Gamma \vdash A : s}{\Gamma; x : A \vdash x : A} \text{ start, } x \notin \Gamma \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma; x : C \vdash A : B} \text{ weakening, } x \notin \Gamma$$

The problem is that if you want to pull some declaration from the middle of an antecedent to the succedent $\Gamma; x : A; \Delta \vdash x : A$ you need to type-check A and whole Δ using the weakening rule. This is not practical and not desirable if you want to prove something like $\Gamma; x : A; \Delta \vdash x : A$ about arbitrary Δ.

Coq has the rule

$$\frac{\mathcal{WF}(E)[\Gamma; x : T; \Delta[x]\ ]}{E[\Gamma; x : T; \Delta[x]\ ] \vdash x : T} \quad (Var)$$

but then you can hardly prove something like $E[\Gamma] \vdash t : T$ for t and T not depending on Γ without assuming $\mathcal{WF}(E)[\Gamma]$. So this kind of assumption would precede any theorem.

We decided to use the following set of rules:

$$\Gamma \vdash \mathsf{Prop} : \mathsf{Type(i)} \qquad \Gamma \vdash \mathsf{Set} : \mathsf{Type(i)} \qquad \Gamma \vdash \mathsf{Type(i)} : \mathsf{Type(j)} \qquad (\text{axioms, } i < j)$$

$$\frac{\Gamma; \Delta \vdash T : s}{\Gamma; x : T; \Delta \vdash x : T} \quad (Var)$$

$$\frac{\Gamma; \Delta \vdash A : B \qquad \Gamma; \Delta \vdash C : s}{\Gamma; x : C; \Delta \vdash A : B} \quad (Weak)$$

So unlike rules in [1] we allow to insert new declarations in the middle of hypotheses list. We also allow nonsense in hypotheses (because of our choice of axioms) but it seems alright - falsum derives anything.

4.2 Lambda Calculus

Implementation of the lambda part of CIC is pretty straightforward, after we settled with treating of $\mathcal{WF}$ and don't tell anything about environment E.

We didn't implement "let-in" construction and definition $x := t : T$ because first of all they seem redundant. Secondly, the majority of the rules do not distinguish definition $x := t : T$ and variable declaration $x : T$, so for now we decided not to complicate our implementation with such a polymorphism.

4.3 Inductive Definitions

Inductive definitions allow us to introduce new types and constructors of these types. $\mathsf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$ is a formal representation of an inductive definition valid in context Γ with parameters Γ_P, a context of definitions Γ_I and a context of constructors Γ_C. Γ_I actually contains types defined by the inductive definition.

Example Parameterized lists is defined as follows:

$$\mathsf{Ind}()[A : \mathsf{Set}](\mathsf{List} : \mathsf{Set} := \mathsf{nil} : \mathsf{List}, \mathsf{cons} : A \to \mathsf{List} \to \mathsf{List})$$

List is a new inductive type, nil (an empty list) and cons (a concatenation of an element and a list) are the constructors of type List; A is a parameter of type Set. List A is a type of lists with elements of type A.

Since Ind has contexts as parameters it has to have flexible arity. As it was mentioned, in MetaPRL the only construct with flexible arity is sequent term. But we should not simply write $\Gamma; \Gamma_P; \Gamma_I; \Gamma_C \vdash \cdot$, because there is no way to tell later which hypotheses are from context Γ, which hypotheses are from context Γ_P, etc. Of course we can reserve special terms to separate those contexts but MetaPRL allows nested sequents so we can write:

$$\mathbf{sequent}\{\Gamma \vdash \mathbf{sequent}\{\Gamma_P \vdash \mathbf{sequent}\{\Gamma_I \vdash \mathbf{sequent}\{\Gamma_C \vdash A\}\}\}\}$$

because we use nested sequents all over the place we label all sequents generously:

$$\begin{array}{l}\mathbf{sequent}\{\Gamma \vdash \\ \quad \mathbf{sequent}[\text{IndParams}]\{\Gamma_P \vdash \\ \quad\quad \mathbf{sequent}[\text{IndTypes}]\{\Gamma_I \vdash \\ \quad\quad\quad \mathbf{sequent}[\text{IndConstrs}]\{\Gamma_C \vdash A\}\}\}\}\end{array}$$

We do not label the outermost sequent because Γ really plays role of hypotheses so outermost sequent is really logical, whereas all other sequents here are merely placeholders with an arbitrary arity. Using display forms we can easily give it a "traditional" format

$$\mathsf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)A$$
or
$$\mathsf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \vdash A$$
or
$$\Gamma \vdash \mathsf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)A$$

which we will use for the rest of the paper. Here A is the actual meaning of the term $\mathsf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)A$ but A can refer to the inductive definition it is wrapped in. Note that due to the nesting, variables declared in an outer sequent's antecedent are bound in all inner sequents but that's exactly what we want.

Types of inductive types and constructors are described by the following two rules. For the rest of the paper we assume that Γ_P is $[p_1 : P_1; \ldots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$.

$$\frac{\mathsf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \in E}{(I_j : (p_1 : P_1) \ldots (p_r : P_r) A_j) \in E} \qquad (j = 1 \ldots k)$$

$$\frac{\mathsf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \in E}{(c_i : (p_1 : P_1) \ldots (p_r : P_r) C_i\{I_j/(I_j p_1 \ldots p_r)\}_{j=1\ldots k}) \in E} \qquad (i = 1 \ldots k)$$

here $(x : S)T$ is a dependent product type (or dependent function type) and it associates to the right.

Aside from giving certain types to inductive types and constructors these rules say that if an inductive definition was given all types and constructors from it are injected in the environment (thus becoming accessible for the later use).

Of course we have to give some explicit meaning for all "..." in those rules and for "massive" simultaneous substitution $I_j/(I_j p_1 \ldots p_r)$. Again we use sequents to express something with flexible arity.

Example We define $(x_1 : T_1) \ldots (x_n : T_n)S$ using two rewrites over sequent term **sequent**[longProduct]$\{x_1 : T_1; \ldots; x_n : T_n \vdash S\}$. For readability we will write $\mathsf{longProduct}\{x_1 : T_1; \ldots; x_n : T_n \vdash S\}$:

$$\mathsf{longProduct}\{\vdash S\} \longleftrightarrow S \qquad \text{base case, } n = 0$$
$$\mathsf{longProduct}\{\Gamma; x : T \vdash S\} \longleftrightarrow \mathsf{longProduct}\{\Gamma \vdash (x : T)S\} \qquad \text{rec. step}$$

on each iteration rightmost declaration $x : T$ is taken from the context Γ and used to form a function type $(x : T)S$ to the result S of the previous iteration.

For the latter rule we need to give definitions of massive application, product and substitution simultaneously because all bindings in the rule have to be preserved correctly. It unfolds to 8 rewrites that act as one recursive function on contexts (basically base case and recursive step for each operation which is 6 already plus some glue).

The next rule tells us if inductive definition is correct.

$$\frac{(E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1\ldots k} \qquad (E[\Gamma; \Gamma_P; \Gamma_I] \vdash C_i : s_{p_i})_{i=1\ldots n}}{\mathcal{WF}(E; \mathsf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C))[\Gamma]}$$

providing the following side conditions hold:

- $k > 0$, I_j, c_i are different names for $j = 1 \ldots k$ and $i = 1 \ldots n$
- for $j = 1 \ldots k$ we have A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$
- for $i = 1 \ldots n$ we have C_i is a type of constructor of I_{p_i} which satisfies the positivity condition for $I_1 \ldots I_k$ and $c_i \notin \Gamma \cup E$

As you can see this rule has a few side conditions. We need to formalize those side conditions via rules and/or rewrites. Side conditions of this rule operate with notions:

- A_j is an arity of sort s_j
- C_i is a type of constructor of I_{p_i}
- C_i satisfies the positivity condition for a constant X
- constant X occurs strictly positively in T

Example The constant X occurs *strictly positively* in T in the following cases:
- X doesn't occur in T
- T converts to $(Xt_1 \dots t_n)$ and X does not occur in any of t_i
- T converts to $(x : U)V$ and X does not occur in type U but occurs strictly positively in type V

actually there is a fourth case but it is too complicated for the discussion.

And the formalization of this definition in MetaPRL looks like this:

$$\frac{}{\Gamma; x : T; \Delta \vdash \mathsf{strictly\_pos}\{x; S\}} \qquad (\textit{base case})$$

here x does not occur freely in S because according to MetaPRL syntax we would have to say $S[x]$ in order to allow free occur

$$\frac{}{\Gamma; x : T; \Delta \vdash \mathsf{strictly\_pos}\{x; \mathsf{appContext}\{\Sigma \vdash x\}\}} \qquad (\textit{application case})$$

here again x does not occur freely in Σ because according to MetaPRL syntax we would have to say $\Sigma[x]$ in order to allow free occur

$$\frac{\Gamma; x : T; \Delta; y : U \vdash \mathit{strictly\_pos}\{x; V[y; x]\}}{\Gamma; x : T; \Delta \vdash \mathsf{strictly\_pos}\{x; y : U \to V[y; x]\}} \qquad (\textit{function case})$$

again x does not occur freely in U.

Because we do not use $\mathcal{WF}$-judgement we need some special treatment for the conclusion of the last rule. We use another judgement

$$\Gamma \vdash \mathsf{IndWF}[\Gamma_P](\Gamma_I := \Gamma_C)$$

which sole purpose is to claim correctness of the inductive definition.

As it was said we do not add types and constructors from inductive definitions to the global environment hence we carry whole inductive definitions everywhere we use it.

Example Using inductive definition of parameterized lists we say:

$$\begin{aligned}
\mathsf{List} &::= \mathsf{Ind}[A : \mathsf{Set}](\mathit{List} : \mathsf{Set} := \mathit{nil} : \mathit{List}; \mathit{cons} : A \to \mathit{List} \to \mathit{List})\mathit{List} \\
\mathsf{nil} &::= \mathsf{Ind}[A : \mathsf{Set}](\mathit{List} : \mathsf{Set} := \mathit{nil} : \mathit{List}; \mathit{cons} : A \to \mathit{List} \to \mathit{List})\mathit{nil} \\
\mathsf{cons} &::= \mathsf{Ind}[A : \mathsf{Set}](\mathit{List} : \mathsf{Set} := \mathit{nil} : \mathit{List}; \mathit{cons} : A \to \mathit{List} \to \mathit{List})\mathit{cons}
\end{aligned}$$

To support this approach our implementation has three rewrites:

$$\mathsf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)t_{\{\}} \leftrightarrow t$$

$$\mathsf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; x : T; \Delta_C)t[x] \leftrightarrow \mathsf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; x : T; \Delta_C) \\ t[\mathsf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; x : T; \Delta_C)x]$$

$$\mathsf{Ind}[\Gamma_P](\Gamma_I; x : T; \Delta_I := \Gamma_C[x])t[x] \leftrightarrow \mathsf{Ind}[\Gamma_P](\Gamma_I; x : T; \Delta_I := \Gamma_C[x]) \\ t[\mathsf{Ind}[\Gamma_P](\Gamma_I; x : T; \Delta_I := \Gamma_C[x])x]$$

The first rewrite says that if term t under inductive definition does not really depend on it, we can get rid of inductive definition and use just t. Second and third rewrite say that any occurrence of inductive type or constructor (under inductive definition) can be wrapped additionally with one more layer of that inductive definition. Having in mind that rewrites are bidirectional we can prove such trivial facts as:

$$\mathsf{List} \in \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{nil} \in (A : \mathsf{Set})(\mathsf{List}A)$$
$$\mathsf{cons} \in (A : \mathsf{Set})(A \to \mathsf{List}A \to \mathsf{List}A)$$

Up to this point we were describing actually working implementation. It includes all the necessary rules, rewrites and tactics for rudimentary proof automation. The example of parameterized lists is proved correct and simple facts given above are proved. The implementation is available for download under GPL license from the MetaPRL CVS server
`http://cvs.metaprl.org:12000/cvsweb/metaprl/theories/cic/`.

4.4 Implementation of Cases and Fixpoint

Besides defining inductive types, establishing their sorts and types of constructors one needs means for case analysis of such types and recursion over inductive types. In CIC (Coq) there are two separate operations - case analysis and recursion (fixpoint) each accompanied with a certain number of rules governing it.

Unfortunately we again face the problem of formalizing side conditions. Consider an inductive definition with several mutually defined types. The case analysis rule has to collect all constructors for one of those types from the list of all constructors of that inductive definition. This was the place where we've got stuck. Although the above condition seems expressible as a collection of rules we don't know any elegant (and efficient) approach. So we decided there is no point in formalizing case analysis and fixpoint rules if it would be too slow and no competitor to Coq.

We do consider an alternative approach. It's possible to wrap each rule in a tactic and implement too complicated side conditions in the tactic. Such tactics will check too intricate syntactical conditions and pre-compute parameters for rules (e.g. extract all appropriate constructors for case analysis rule). Those tactics should be considered as a part of the trusted core but we will get much better efficiency. Such an implementation would be no less reliable than Coq

because (as far as we understand) in Coq this logic is also hard-coded and not explicitly written as a system of rules.

5 Future work

Presently we are at the crossroad of several treatments for the case-analysis and fixpoint rules, which are:

- Find a way to represent side conditions of those rules as rules and rewrites. This will most likely lead to a significant drop in the speed comparing with Coq but MetaPRL trusted core won't be extended.
- Wrap each rule in a tactic and implement too complicated side conditions in the tactic. This would probably boost the performance. But such tactics would actually extend MetaPRL trusted core.
- Find a formal generic notation that would allow to implement case-analysis and fixpoint rules nicely. If successful this might be a good tradeoff between performance and extension of the trusted core. And we would got an extra bonus - improve the expressiveness of the MetaPRL meta-language.

After the decision is made the rest of the CIC core and basic proof automation will be implemented. Then we will benchmark our implementation against Coq. If successful, more steps towards compatibility with the existing Coq-libraries will be made. The ultimate goal is to support import or direct access to Coq library files.

References

1. Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.
2. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.
3. Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
4. Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, Berlin, 1990.
5. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

6. Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.
7. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
8. INRIA. *The Coq Proof Assistant Reference Manual*, 2003.
9. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.
10. Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

Towards Verified Virtual Memory in L4

Gerwin Klein and Harvey Tuch

[1] University of New South Wales, Sydney 2052, Australia
[2] National ICT Australia*, Sydney, Australia
{gerwin.klein|harvey.tuch}@nicta.com.au

Abstract. We report on the initial stage of an on-going verification project: the formalisation and verification of the L4 μ-kernel. We describe an abstract model of the virtual memory subsystem in L4, prove safety properties about this model, and describe refinement of the abstract model towards the implementation of L4. All formalisations and proofs have been carried out in the theorem prover Isabelle.

1 Introduction

L4 is a second generation microkernel based on the principles of minimality, flexibility, and efficiency [10]. It provides the traditional advantages of the microkernel approach to system structure, namely improved reliability and flexibility, while overcoming the performance limitations of the previous generation of microkernels. With implementation sizes in the order of 10,000 lines of C++ and assembler code it is about an order of magnitude smaller than Mach and two orders of magnitude smaller than Linux.

The operating system (OS) is clearly one of the most fundamental components of non-trivial systems. The correctness and reliability of the system critically depends on the OS. In terms of security, the OS is part of the trusted computing base, that is, the hardware and software necessary for the enforcement of a system's security policy. It has been repeatedly demonstrated that current operating systems fail at these requirements of correctness, reliability, and security. Microkernels address this problem by applying the principles of minimality and least privilege to operating system architecture. However, the success of this approach is still predicated on the microkernel being designed and implemented correctly. We can address this by formally modelling and verifying it.

L4 has a design that is not only geared towards flexibility and reliability, but is of a size which makes formalisation and verification feasible. Compared to other operating system kernels, L4 is very small; compared to the size of other verification efforts, 10,000 lines of code is still considered a very large and complex system. Our methodology for solving this verification problem is shown in Fig. 1. It is a classic refinement strategy. We start out from an abstract model of the kernel that is phrased in terms of user concepts as they are explained in

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

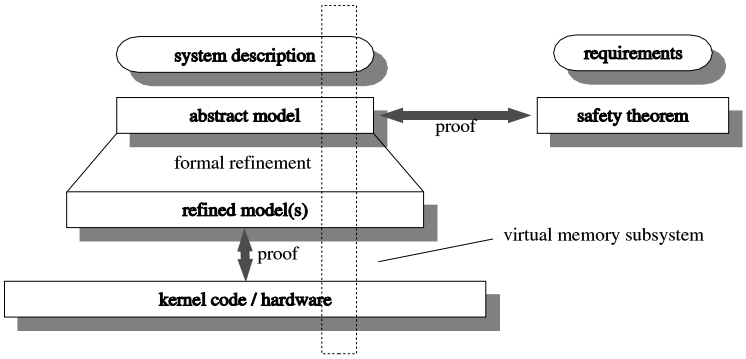


Fig. 1. Overview

the L4 reference manual [1]. This is the level at which most of the safety and security theorems will be shown. We then formally refine this abstract model in multiple property preserving steps towards the implementation of L4. The last step consists of verifying that the C++ and assembler source code of the kernel correctly implements the most concrete refinement level. At the end of this process, we will have shown that the kernel source code satisfies the safety and security properties we have proved about the abstract model.

To keep complexity and time manageable, we have decided to take a thin vertical slice out of this refinement process and to test the methodology on one non-trivial subsystem of the kernel initially. This will not give hard safety guarantees about the full system, but it will increase confidence in the implementation and improve understanding of the target subsystem. The goal is to move through the full process quickly and to uncover problems in the interaction of refinement layers and the different formalisms utilised.

In this paper we report on first experiences with this project. L4 provides three main abstractions: threads, address spaces, and inter-process communication (IPC). We have chosen to start with address spaces. This is supported by the virtual memory subsystem of the kernel and is fundamental for implementing separation and security policies on top of L4. We have built an abstract model of address spaces and we show a first refinement of it.

One of the central questions in any verification project is: When exactly is the specification of the system correct? What is the system supposed to do? In this case we have taken the L4 X.2 API description as the main reference [1] and use the L4Ka::Pistachio [8] implementation on the ARM architecture to resolve ambiguities and address implementation issues, in addition to discussions with the developers on the *pistachio-core* mailing list.

As we are mainly trying to test the methodology, we are making some simplifying assumptions in the formalisation. We are also not planning to verify the current implementation of L4Ka::Pistachio. On the contrary, it is a goal and expected outcome of this project that we clarify and simplify the implementation.

If verification makes it necessary, even a complete reimplementation of the L4 X.2 API is possible.

Earlier work on operating system kernel formalisation and verification includes PSOS [11] and UCLA Secure Unix [15]. The focus of this work was on capability-based security kernels, allowing security policies such as multi-level security to be enforced. These efforts were hampered by the lack of mechanisation and appropriate tools available at the time and so while the designs were formalised, the full verification proofs were not practical. Later work, such as KIT [3], describes verification of properties such as process isolation to source or object level but with kernels providing far simpler and less general abstractions than modern microkernels. There exists some work in the literature on the modelling of microkernels at the abstract level with varying degrees of completeness. Bevier and Smith [4] specify legal Mach states and describe Mach system calls using temporal logic. Shapiro and Weber [13] give an operational semantics for EROS and prove a confinement security policy. Our work differs in that we plan to formally relate our model to the implementation. Some case studies [7,5,14] appear in the literature in which the IPC and scheduling subsystems of microkernels have been described in PROMELA and verified with the SPIN model checker. These abstractions were not necessarily sound, having been manually constructed from the implementations, and so while useful for discovering concurrency bugs do not provide guarantees of correctness. Finally, the VFiasco project, working with the Fiasco implementation of L4, has published exploratory work on the issues involved in C++ verification at the source level [9].

After introducing our notation in the following section, we first present an abstract conceptual model of virtual memory in L4 in section 3 and refine it towards an implementation in section 4.

2 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

$$\textbf{datatype}\ 'a\ option = None \mid Some\ 'a$$

adjoins a new element *None* to a type $'a$. For succinctness we write $\lfloor a \rfloor$ instead of *Some a*.

Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$.

Partial functions are modelled as functions of type $'a \Rightarrow 'b\ option$, where *None* represents undefinedness and $f\,x = \lfloor y \rfloor$ means x is mapped to y. We call such functions *maps*, and abbreviate $f(x{:=}\lfloor y \rfloor)$ to $f(x \mapsto y)$. The map $\lambda x.\ None$ is written *empty*, and *empty*(...), where ... are updates, abbreviates to [...]. For example, $empty(x{\mapsto}y)$ becomes $[x \mapsto y]$.

Implication is denoted by $\Longrightarrow$ and $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

Finally, how are the formulae you see related to the formal Isabelle text? Our motto is

What you see is what we proved!

Isabelle theories can be augmented with LaTeX text which may contain references to Isabelle theorems (by name — see chapter 4 of [12]). We use this presentation mechanism to generate the text for most of the definitions and all of the theorems in this paper automatically.

3 Abstract Address Space Model

The virtual memory subsystem in L4 provides a flexible, hierarchical way of manipulating the mapping from virtual to physical memory pages of address spaces at user-level. We now present a formal model for this. Although the granularity at which L4 maps memory is the page level and does not go down to single addresses, we use the terms *address* and *page* interchangeably in the following.

3.1 Address Spaces

Fig. 2 illustrates the concept of hierarchical mappings. Large boxes depict virtual address spaces. The smaller boxes inside stand for virtual pages in the address space. The rounded box at the bottom is the set of physical pages. The arrows stand for direct mappings which connect pages in one address spaces to addresses in (possibly) other address spaces. In well-behaved states, the transitive closure of mappings always ends in physical pages. The example in Fig. 2 maps virtual page v_1 in space n_1, as well as v_2 in n_2, and v_4 in n_4 to the physical page r_1.

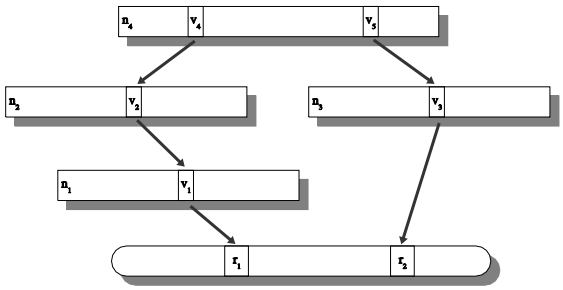


Fig. 2. Address Spaces

Formally, we use the types R for the physical pages (r_1, r_2, etc.), V for virtual pages (v_1, v_2, etc.), and N for the names of address spaces (n_1, n_2, etc.).

A position in this picture is determined uniquely by either naming a virtual page in a virtual address space, or by naming a physical page. We call these the mappings M:

$$\textbf{datatype}\ M = \mathit{Virtual}\ N\ V \mid \mathit{Real}\ R$$

An address space associates with each virtual page either a mapping, or nothing (the nil page). We implement this in Isabelle by the *option* datatype:

$$\textbf{types}\ \mathit{space} = V \Rightarrow M\ \mathit{option}$$

The machine state is then a map from address space names to address spaces. Not all names need to be associated with an address space, so we use *option* again:

$$\textbf{types}\ \mathit{state} = N \Rightarrow \mathit{space}\ \mathit{option}$$

To relate these functions to the arrows in Fig. 2, we use the concept of *paths*. The term $s \vdash x \rightsquigarrow^1 y$ means that in state s there is a direct path from position x to position y. There is a direct path from position $\mathit{Virtual}\ n\ v$ to another position y if in state s the address space with name n is defined and maps the virtual page v to y. There can be no paths starting at physical pages. Formally,

$$s \vdash x \rightsquigarrow^1 y = (\exists n\ v\ \sigma.\ x = \mathit{Virtual}\ n\ v \wedge s\ n = \lfloor\sigma\rfloor \wedge \sigma\ v = \lfloor y\rfloor)$$

We write $\_ \vdash \_ \rightsquigarrow^+ \_$ for the transitive and $\_ \vdash \_ \rightsquigarrow^* \_$ for the reflexive and transitive closure of the direct path relation.

3.2 Operations

The L4 kernel exports the following basic operations on address spaces: *unmap*, *flush*, *map*, and *grant*. The former two operations remove mappings, the latter two create or move mappings. We explain and define them below.

Fig. 3 illustrates the $\mathit{unmap}\ n\ v$ operation. It is the most fundamental of the operations above. We say a space n unmaps v if it removes all mappings that depend on $\mathit{Virtual}\ n\ v$, or in terms of paths if it removes all edges leading to $\mathit{Virtual}\ n\ v$.

To implement this, we use a function *clear* that, given name n, page v, and address space σ in a state s, returns σ with all v' leading to $\mathit{Virtual}\ n\ v$ mapped to *None*.

$\mathit{clear} :: N \Rightarrow V \Rightarrow \mathit{state} \Rightarrow \mathit{space} \Rightarrow \mathit{space}$
$\mathit{clear}\ n\ v\ s\ \sigma \equiv$
$\lambda v'.\ \mathsf{case}\ \sigma\ v'\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None}$
$\quad \mid \lfloor m\rfloor \Rightarrow \mathit{if}\ s \vdash m \rightsquigarrow^*\ \mathit{Virtual}\ n\ v\ \mathsf{then}\ \mathit{None}\ \mathsf{else}\ \lfloor m\rfloor$

An $\mathit{unmap}\ n\ v$ in state s then produces a new state in which each address space is cleared of all paths leading to $\mathit{Virtual}\ n\ v$.

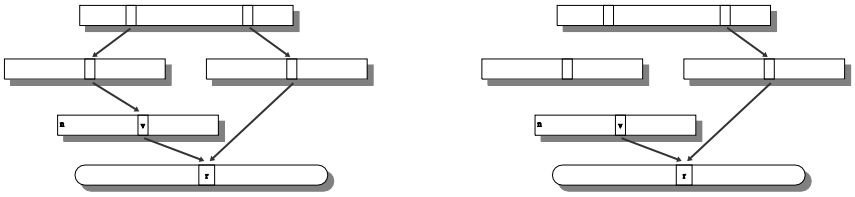


Fig. 3. The *unmap* operation (before and after)

$unmap :: N \Rightarrow V \Rightarrow state \Rightarrow state$
$unmap\ n\ v\ s \equiv \lambda n'.\ \mathsf{case}\ s\ n'\ \mathsf{of}\ None \Rightarrow None \mid \lfloor\sigma\rfloor \Rightarrow \lfloor clear\ n\ v\ s\ \sigma\rfloor$

For updating a space with name n at page v with a new mapping m we write $n,v \leftarrow m$, where m may be *None*.

$n,v \leftarrow m \equiv \lambda s.\ s(n := \mathsf{case}\ s\ n\ \mathsf{of}\ None \Rightarrow None \mid \lfloor\sigma\rfloor \Rightarrow \lfloor\sigma(v := m)\rfloor)$

With this, the flush operation is simply *unmap* followed by setting n,v to *None*.

$flush :: N \Rightarrow V \Rightarrow state \Rightarrow state$
$flush\ n\ v \equiv n,v \leftarrow None \circ unmap\ n\ v$

The remaining two operations *map* and *grant* establish new mappings in the receiving address space. To ensure a consistent new state, this new mapping must ultimately be connected to a physical page. We call a mapping m *valid* in state s (written $s \vdash m$) if it is a physical page, or if it is of the form *Virtual* $n\ v$ and is the source of some direct path. We show later that in all reachable states of the system, this definition is equivalent to saying that the mapping leads to a physical page.

$s \vdash m \equiv \mathsf{case}\ m\ \mathsf{of}\ Virtual\ n\ v \Rightarrow \exists x.\ s \vdash m \rightsquigarrow^1 x \mid Real\ r \Rightarrow True$

Before the kernel establishes a new value, the destination is always flushed. This may invalidate the source. The operation only continues if the source is still valid, otherwise it stops. We capture this behaviour in a slightly modified update notation $\leftharpoonup$:

$n,v \leftharpoonup m \equiv \lambda s.\ \mathsf{let}\ s_0 = flush\ n\ v\ s\ \mathsf{in}\ (\mathsf{if}\ s_0 \vdash m\ \mathsf{then}\ n,v \leftarrow \lfloor m\rfloor\ \mathsf{else}\ id)\ s_0$

In L4, an address space n can *map* a page v to another space n' at page v'. Again, the operation only goes ahead, if the mapping *Virtual* $n\ v$ is valid:

$map :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$map\ n\ v\ n'\ v'\ s \equiv \mathsf{if}\ \neg\ s \vdash Virtual\ n\ v\ \mathsf{then}\ s\ \mathsf{else}\ (n',v' \leftharpoonup Virtual\ n\ v)\ s$

Fig. 4 shows an example for the *map* operation. Address space n maps page v to n' at v'. The destination n',v' is first flushed and then updated with the new mapping *Virtual* $n\ v$.

A space n can also *grant* a page v to v' in n'. As illustrated in Fig. 5, granting updates n',v' to the value of n at v and flushes the source n,v.

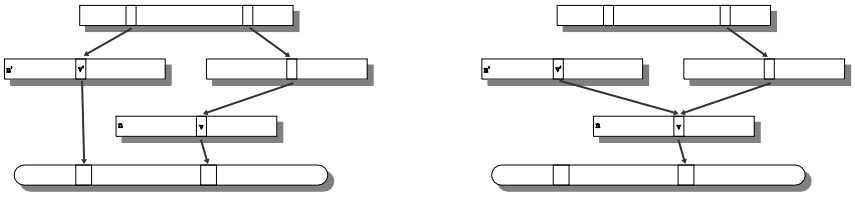


Fig. 4. The *map* operation (before and after)

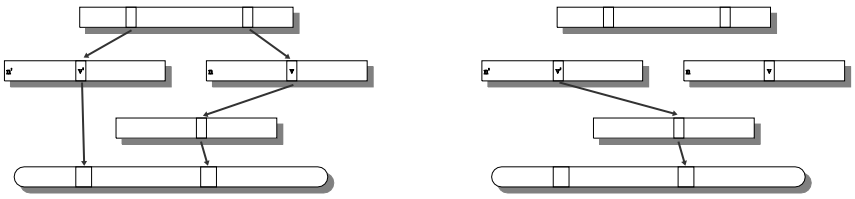


Fig. 5. The *grant* operation (before and after)

$grant :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$grant\ n\ v\ n'\ v'\ s \equiv$
$if\ \neg\ s \vdash Virtual\ n\ v\ then\ s$
$else\ let\ \lfloor\sigma\rfloor = s\ n;\ \lfloor m\rfloor = \sigma\ v\ in\ (flush\ n\ v \circ n',v' \leftharpoonup m)\ s$

This concludes the kernel operations on address spaces. We have also modelled the hardware memory management unit (MMU). On this abstract level, all the MMU does is lookup: it determines which physical page needs to be accessed for each virtual page v and address space n. We write $s \vdash n\ v \rhd r$ if lookup of page v in the address space with name n in state s yields the physical page r. As we already have the concepts of paths, this is easily described formally:

$s \vdash n,v \rhd \lfloor r\rfloor = s \vdash Virtual\ n\ v \rightsquigarrow^+ Real\ r$
$s \vdash n,v \rhd None = (\exists\sigma.\ s\ n = \lfloor\sigma\rfloor \wedge \sigma\ v = None) \vee s\ n = None$

The model in this section is based on an earlier pen-and-paper formalisation of L4 address spaces by Liedtke [10]. Formalising it in Isabelle/HOL eliminated problems like the mutual recursive definition of the update and flush functions being not well-founded. It would be well-founded—at least on reachable kernel states—if the model had the property that no loops can be constructed in address spaces. This is not true in the original model. The operation $map\ n\ v\ n'\ v'$ followed by $grant\ n'\ v'\ n\ v$ is a counter example. We also have introduced the formal concept of valid mappings to establish this no-loops property as well as the fact that any page that is mapped at all is mapped to a physical address.

3.3 An abstract data type for virtual memory

In the following we phrase the model of virtual memory and of the MMU hardware in terms of an abstract data type consisting of the type *state* and the operations detailed above. This data type (not to be confused with Isabelle's keyword **datatype**) is used implicitly by any user-level program. Even if the program does not invoke any mapping operations directly, the CPU performs a lookup operation with every memory access.

Putting the operations in terms of an abstract data type enables us to formulate refinement explicitly: if the data type of the abstract address spaces model is replaced with the data type of more concrete models (and finally the implementation) the program will not have any observable differences in behaviour.

Formally we define an abstract data type as a record consisting of an initial set of states and of a transition relation that models execution:

$$\begin{array}{l}\textbf{record}\ ('a,\ 'j)\ \mathit{DataType} = \\ \mathit{Init} :: 'a\ \mathit{set} \\ \mathit{Step} :: 'j \to ('a \times 'a)\ \mathit{set}\end{array}$$

For our virtual memory model, the operations are enumerated in the index type *VMIndex*:

$$\begin{array}{l}\textbf{datatype}\ \mathit{VMIndex} = \mathsf{create}\ N \mid \mathsf{unmap}\ N\ V \mid \mathsf{flush}\ N\ V \mid \mathsf{map}\ N\ V\ N\ V \\ \qquad\qquad \mid \mathsf{grant}\ N\ V\ N\ V \mid \mathsf{lookup}\ N\ V\ (R\ \mathit{option})\end{array}$$

The definition of the abstract model $\mathcal{A}$ in terms of a data type is then:

$\mathit{Init}\ \mathcal{A} = \{[\sigma_0 \mapsto \sigma]\ |\sigma.\ \mathit{inj}_p\ \sigma \wedge \mathit{ran}\ \sigma \subseteq \mathit{range}\ \mathit{Real}\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{lookup}\ n\ v\ r) = \{(s,\ s') \mid s = s' \wedge s \vdash n,v \rhd r\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{create}\ n) = \{(s,\ s') \mid s\ n = \mathit{None} \wedge s' = s(n \mapsto \mathit{empty})\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{unmap}\ n\ v) = \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s' = \mathit{unmap}\ n\ v\ s\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{flush}\ n\ v) = \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s' = \mathit{flush}\ n\ v\ s\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{map}\ n\ v\ n'\ v') = \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s\ n' \neq \mathit{None} \wedge s' = \mathit{map}\ n\ v\ n'\ v'\ s\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{grant}\ n\ v\ n'\ v') =$
$\{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s\ n' \neq \mathit{None} \wedge s' = \mathit{grant}\ n\ v\ n'\ v'\ s\}$

The boot process creates an address space σ_0 that is an injective mapping from virtual to physical pages. The functions *ran* and *range* return the codomain of a function, where *ran* works on functions $'a \Rightarrow 'b\ \mathit{option}$ and *range* on total functions. Injectivity is constrained to the part of the function that returns $\lfloor x \rfloor$: $\mathit{inj}_p\ f \equiv \mathit{inj\text{-}on}\ f\ \{x \mid \exists y.\ f\ x = \lfloor y \rfloor\}$.

The lookup operation is special. In the context of a real system this operation would return a value, since one of the points of the virtual memory abstraction is to provide address translation. If a lookup yields a *None* result the kernel typically raises a *page fault* exception. Since we do not model the larger system, we simplify lookup instead to a subset of the identity relation on *state*.

Creating a new address space n is modelled by updating the state s at n with the predefined map *empty*. The other mapping operations have been defined above. All of them require the address spaces they operate on to be valid. This condition is ensured automatically in the current L4 implementation as the address spaces are determined by sender and receiver of an IPC operation.

3.4 Properties

We have shown a number of safety properties about the abstract address space model. They are formulated as invariants over the abstract datatype. A set of states I is an invariant if it contains all initial states and if execution of any operation in a state of I again leads to a state in I. We write $\mathcal{D} \models I$ when I is an invariant of data type $\mathcal{D}$.

Theorem 1. *There are no loops in the address space structure.*

$$\mathcal{A} \models \{s \mid \forall x.\ \neg\ s \vdash x \rightsquigarrow^{+} x\}$$

The proof is by case distinction on the operations and proceeds by observing how each operation changes existing paths. Theorem 1 is significant for implementing the lookup function efficiently. It also ensures that internal kernel functions can walk the corresponding data structures naively. Together with the properties below it says that address spaces always have a tree structure.

Theorem 2. *All valid pages translate to physical pages.*

$$\mathcal{A} \models \{s \mid \forall x.\ s \vdash x \longrightarrow (\exists r.\ s \vdash x \rightsquigarrow^{*} \mathit{Real}\ r)\}$$

The proof is again by case distinction on the operations. Together with the following theorem we obtain that address lookup is a total function on data type $\mathcal{A}$.

Theorem 3. *The lookup relation is a function.*

$$[\![s \vdash n{,}v \rhd r;\ s \vdash n{,}v \rhd r']\!] \Longrightarrow r = r'$$

This theorem follows directly from the fact that paths are built on functions.

That address lookup is a total function may sound like merely a nice formal property, but it is quite literally an important safety property in reality. Undefined behaviour, possibly physical damage, may result if two conflicting TLB entries are present for the same virtual address. The current ARM reference manual [2, p. B3-26] explicitly warns against this scenario.

3.5 Simplifications and Assumptions

The current model makes the following simplifications and assumptions.

- The L4Ka::Pistachio API stipulates two regions per address space that are shared between the user and kernel, the *kernel interface page* (KIP) and *user thread control blocks* (UTCBs). These should have a valid translation from virtual to physical memory pages, but can not be manipulated by the mapping operations.
- The mapping operations in L4 work on regions of the address space rather than individual pages. These regions, known as *flexpages*, are $2^k b, k \geq 0$ aligned and sized where b is the minimum page size on the architecture. This introduces significant complexity in the implementation and has a number of

boundary conditions of interest, so adding this to the abstract model would be beneficial. At the same time, it is possible to create systems using L4 that only use the minimum flexpage size so this omission does not pose a serious limitation to the utility of the model.
- *map* and *grant* are implemented through the IPC primitives in L4 and involve an agreement on the region to be transferred between sender and receiver. This can be added when the IPC abstraction is modelled.
- Flexpages also have associated read, write and execute access rights. At present the model can be considered as providing an all or nothing view of access rights.
- We assume that all of the mapping operations are atomic, which is the case in the current non-preemptable implementation, and a single processor, hence a sequential system.

4 Model Refinement

The model in the previous section provides an abstract model of address spaces in L4 but does not bear much resemblance to the kernel implementation. This is not surprising since the kernel must provide an efficient realisation of the mapping operations and the code supporting this executes under time and space restrictions. For the purpose of source-code verification it is desirable to have a more concrete model of the implementation. This model will be more complex and detailed than the previous model and hence less suited to proving properties such as the absence of loops in paths. By showing the concrete model to be a refinement of the abstract model it is possible to retain the ability to reason and prove properties at the abstract level. In this section we provide a motivation and overview of the implementation in the L4Ka::Pistachio kernel of address spaces, and then describe the refinement of the abstract address spaces model.

4.1 L4Ka::Pistachio Implementation

The implementation of address spaces is provided by the hardware and OS virtual memory mechanisms. The lookup relation corresponds to the virtual-to-physical mapping function provided by the MMU on the CPU. This translation is carried out on every memory access and so is critical to system performance. This is typically hidden in the pipeline by an associative cache, called the *translation-lookaside buffer* (TLB), holding a subset of mappings from the *page table* data structure which is located in memory. On a TLB miss the page table is accessed to perform address translation by a hardware mechanism (on the ARM architecture) that walks the page table data structure. The page table must support fast address translation, since TLB misses are frequent enough to warrant this, but this must be balanced with space considerations. In L4Ka::Pistachio a multi-level hierarchical page table is implemented, of which the ARM hardware

defined page table format, a two-level page table, is an instance. The operations that update mappings must also maintain coherence between the TLB and page table, and also the data and instruction caches and memory on ARM since the caches are virtually-addressed.

In addition to the virtual-to-physical mappings, an implementation of L4 address spaces requires a representation of the mappings between address spaces, the *mapping database* (MDB). This is conceptually quite similar to the abstract model, with paths reversed to give a tree rooted at each physical memory page. The *map*, *grant* and *unmap* operations correspond to system calls and execute with a small, fixed-size kernel stack. Hence it is desirable to avoid recursion. This is achieved in L4Ka::Pistachio by implementing the mapping tree with a linked-list representing the preorder traversal of the tree, augmented with depth information. The list is doubly-linked and there are pointers stored between nodes in the mapping database and nodes associated with the corresponding page table nodes to avoid unnecessary traversals of either data structure in the mapping operations.

4.2 Tree Address Space Model

We first show that a model of address spaces with the mapping database as a forest to be a refinement of the model in Section 3. This is a conceptual step. It is the view that most people working with the kernel implementation adopt.

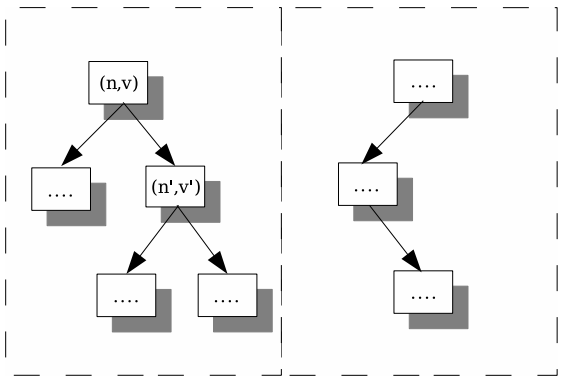


Fig. 6. Forest

$$\textbf{types}\ MDB = (N \times V) \rightharpoonup (N \times V)\ set$$

A tree here is a partial function from a node to a set of child nodes (see Fig. 6). The function is required to be partial so that nodes with no children and nodes not present in the tree can be distinguished.

$$\begin{array}{l}\textbf{record}\ state_1 = \\ N :: N\ set \\ M :: R \Rightarrow MDB\end{array}$$

The N component of the state now contains the names of the valid address spaces and each physical memory page has an associated mapping tree (possibly empty) in the M component of the state.

The direct path relation is defined as

$$s \vdash a \rightsquigarrow^1_1 b = (\exists r\ mn.\ M\ s\ r\ a = \lfloor mn \rfloor \wedge b \in mn)$$

A direct path exists between nodes a and b if b is a child of a in a tree r.

Again, we write $\_ \vdash \_ \rightsquigarrow^+_1 \_$ for the transitive and $\_ \vdash \_ \rightsquigarrow^*_1 \_$ for the reflexive and transitive closure of the direct path relation. A path between a and b indicates that b is in the subtree of a.

Lookup in the tree model is written as $s \vdash n,v \rhd_1 r$ and is defined with:

$s \vdash n,v \rhd_1 \lfloor r \rfloor = (M\ s\ r\ (n,\ v) \neq None)$
$s \vdash n,v \rhd_1 None = ((\forall r.\ M\ s\ r\ (n,\ v) = None) \wedge n \in N\ s \vee n \notin N\ s)$

Lookup corresponds to tree membership for a node.

The $unmap_1$ operation then simply removes all nodes in the subtree of the target from the tree, except the target, and all references to these nodes from other nodes. The notation $s(\!|M := x|\!)$ denotes update of field M in record s with value x.

$unmap_1\ n\ v\ s \equiv$
$s(\!|M := \lambda r\ x.\ \mathsf{case}\ M\ s\ r\ x\ \mathsf{of}\ None \Rightarrow None$
$\quad | \lfloor mn \rfloor \Rightarrow$
$\quad\quad \mathsf{if}\ s \vdash (n,\ v) \rightsquigarrow^+_1 x\ \mathsf{then}\ None$
$\quad\quad \mathsf{else}\ \lfloor \{b \mid b \in mn \wedge \neg\ s \vdash (n,\ v) \rightsquigarrow^+_1 b\} \rfloor |\!)$

Similarly, $flush_1$ removes all nodes in the subtree along with their corresponding references.

$flush_1\ n\ v\ s \equiv$
$s(\!|M := \lambda r\ x.\ \mathsf{case}\ M\ s\ r\ x\ \mathsf{of}\ None \Rightarrow None$
$\quad | \lfloor mn \rfloor \Rightarrow$
$\quad\quad \mathsf{if}\ s \vdash (n,\ v) \rightsquigarrow^*_1 x\ \mathsf{then}\ None$
$\quad\quad \mathsf{else}\ \lfloor \{b \mid b \in mn \wedge \neg\ s \vdash (n,\ v) \rightsquigarrow^*_1 b\} \rfloor |\!)$

map_1 is implemented by inserting a new node for the map destination in the tree beneath the map source.

$map_1\ n\ v\ n'\ v'\ s \equiv$
$\mathsf{if}\ s \vdash n,v \rhd_1 None\ \mathsf{then}\ s$
$\mathsf{else}\ \mathsf{let}\ s' = flush_1\ n'\ v'\ s$
$\quad \mathsf{in}\ \mathsf{if}\ s' \vdash n,v \rhd_1 None\ \mathsf{then}\ s'\ \mathsf{else}\ update\text{-}map_1\ n\ v\ n'\ v'\ s'$

$update\text{-}map_1\ n\ v\ n'\ v'\ s \equiv$
$s(\!|M := \lambda r.\ \mathsf{case}\ M\ s\ r\ (n,\ v)\ \mathsf{of}\ None \Rightarrow M\ s\ r$
$\quad | \lfloor mn \rfloor \Rightarrow M\ s\ r((n,\ v) \mapsto mn \cup \{(n',\ v')\},\ (n',\ v') \mapsto \{\})|\!)$

In $grant_1$ a node is inserted into the tree for the destination if the prior flush and unmap do not result in the source being removed, and any references to the source are replaced by references to the destination node.

$$
\begin{array}{l}
grant_1\ n\ v\ n'\ v'\ s \equiv \\
\textit{if}\ s \vdash n{,}v \rhd_1 None\ \textit{then}\ s \\
\textit{else let}\ s' = flush_1\ n'\ v'\ s \\
\quad \textit{in if}\ s' \vdash n{,}v \rhd_1 None\ \textit{then}\ s'\ \textit{else}\ update\text{-}grant_1\ n\ v\ n'\ v'\ s'
\end{array}
$$

$$
\begin{array}{l}
update\text{-}grant_1\ n\ v\ n'\ v'\ s \equiv \\
\textit{let}\ s' = unmap_1\ n\ v\ s \\
\textit{in}\ s'(\!|M := \lambda r\ x.\ \textit{if}\ x = (n',\ v') \wedge s' \vdash n{,}v \rhd_1 \lfloor r \rfloor\ \textit{then}\ \lfloor\{\}\rfloor \\
\qquad\qquad \textit{else case}\ M\ s'\ r\ x\ \textit{of}\ None \Rightarrow None \\
\qquad\qquad\quad \mid \lfloor mn \rfloor \Rightarrow \\
\qquad\qquad\qquad \textit{if}\ x = (n,\ v)\ \textit{then}\ None \\
\qquad\qquad\qquad \textit{else}\ \lfloor\{b \mid b \in mn \wedge b \neq (n,\ v) \vee \\
\qquad\qquad\qquad\qquad\qquad (n,\ v) \in mn \wedge b = (n',\ v')\}\rfloor\ |\!)
\end{array}
$$

4.3 Refinement Proof

In this section we again phrase the model presented above in terms of a data type. The tree data type $\mathcal{M}$ is:

$$
\begin{array}{l}
Init\ \mathcal{M} = \\
\{(\!|N = \{\sigma_0\},\ M = \lambda r\ nv.\ \textit{if}\ P'\ nv = \lfloor r \rfloor\ \textit{then}\ \lfloor\{\}\rfloor\ \textit{else}\ None|\!)\ |P'. \\
\ inj_p\ P' \wedge fst\ `\ dom\ P' \subseteq \{\sigma_0\}\} \\
Step\ \mathcal{M}\ (\mathsf{lookup}\ n\ v\ r) = \{(s,\ s') \mid s = s' \wedge s \vdash n{,}v \rhd_1 r\} \\
Step\ \mathcal{M}\ (\mathsf{create}\ n) = \{(s,\ s') \mid n \notin N\ s \wedge s' = s(\!|N := insert\ n\ (N\ s)|\!)\} \\
Step\ \mathcal{M}\ (\mathsf{unmap}\ n\ v) = \{(s,\ s') \mid n \in N\ s \wedge s' = unmap_1\ n\ v\ s\} \\
Step\ \mathcal{M}\ (\mathsf{flush}\ n\ v) = \{(s,\ s') \mid n \in N\ s \wedge s' = flush_1\ n\ v\ s\} \\
Step\ \mathcal{M}\ (\mathsf{map}\ n\ v\ n'\ v') = \{(s,\ s') \mid n \in N\ s \wedge n' \in N\ s \wedge s' = map_1\ n\ v\ n'\ v'\ s\} \\
Step\ \mathcal{M}\ (\mathsf{grant}\ n\ v\ n'\ v') = \{(s,\ s') \mid n \in N\ s \wedge n' \in N\ s \wedge s' = grant_1\ n\ v\ n'\ v'\ s\}
\end{array}
$$

We show that the tree data type is a refinement of the abstract data type. Here refinement is taken to mean *data refinement* [6] and we use the proof technique of simulation.

We begin with the abstraction relation R_1 between concrete state s_c and abstract states s_a:

$$
\begin{array}{l}
R_1 \equiv \\
\{(s_c,\ s_a) \mid dom\ s_a = N\ s_c \wedge \\
\qquad (\forall n\ v\ r.\ s_a \vdash n{,}v \rhd \lfloor r \rfloor = s_c \vdash n{,}v \rhd_1 \lfloor r \rfloor) \wedge \\
\qquad (\forall n\ v\ n'\ v'.\ s_a \vdash Virtual\ n\ v \rightsquigarrow^1 Virtual\ n'\ v' = s_c \vdash (n',\ v') \rightsquigarrow^1{}_1\ (n,\ v))\}
\end{array}
$$

Here it is clear that the path relation in the tree model is the inverse of the path relation in the abstract model.

We then show that the diagrams in Fig. 7 commute, for all operations. This is achieved by showing forward simulation:

$$C \leq_F A \equiv \exists r.\ Init\ C \subseteq r\ ``\ Init\ A \wedge (\forall j.\ r\ ;;\ Step\ C\ j \subseteq Step\ A\ j\ ;;\ r)$$

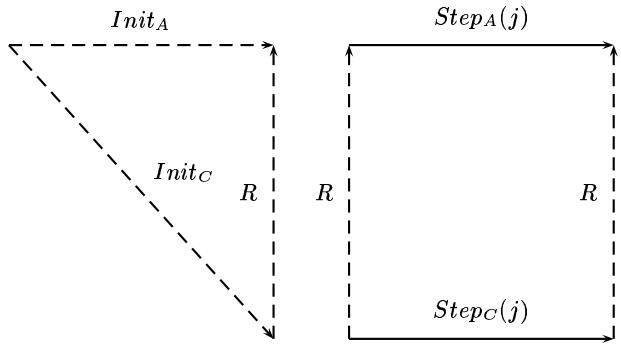


Fig. 7. Simulation

where " is the image of a set under a relation, and ;; the composition of two relations.

Theorem 4. *The tree data type simulates the abstract data type*

$$\mathcal{M} \leq_F \mathcal{A}$$

The proof is by case distinction on the operations of the data type. It proceeds by observing how each operation changes the state in terms of the path and lookup relations on the concrete and abstract level. For example, the direct path relation after flush can be shown to be:

$$\mathit{flush}\ n\ v\ s \vdash x \rightsquigarrow^1 y = (s \vdash x \rightsquigarrow^1 y \wedge \neg\, s \vdash x \rightsquigarrow^* \mathit{Virtual}\ n\ v)$$

$$\mathit{flush}_1\ n\ v\ s \vdash x \rightsquigarrow^1{}_1\ y = (s \vdash x \rightsquigarrow^1{}_1\ y \wedge \neg\, s \vdash (n,\ v) \rightsquigarrow^*{}_1\ y)$$

Simulation gives that the properties proved as invariants on the abstract data type also hold on the concrete data type, i.e. the safety properties proved in Section 3.4 also hold on the concrete data type.

Also, since the operations are deterministic, the simulation also holds in the other direction.

Theorem 5. *The abstract data type simulates the tree data type*

$$\mathcal{A} \leq_F \mathcal{M}$$

4.4 Further Refinement

The next step in the refinement process is to implement the forest with a list model. The state space for this is based on the following type:

record *TreeListNode* =
Next :: *TreeListNodeName option*
Prev :: *TreeListNodeName option*
PTE :: *PTEName*
Depth :: *nat*

$$\begin{array}{l}\textbf{record}\ \mathit{TreeListHeap} = \\ \mathit{Valid} :: \mathit{TreeListNodeName}\ \mathit{set} \\ \mathit{Heap} :: \mathit{TreeListNodeName} \Rightarrow \mathit{TreeListNode}\end{array}$$

where *TreeListNodeName* and *PTEName* are uninterpreted types. These represent pointers to list nodes and page table entries respectively.

The mapping operations in this model are closer to those in the implementation. Unmap/flush iterate over the subtree unlinking nodes, map inserts a node into the list immediately after the destination node and grant replaces the source node with that of the destination in the list.

The following subtree relation can be used to connect the list to the tree model.

$$s \vdash x \mapsto y = (\mathit{Next}\ (\mathit{Heap}\ s\ x) = \lfloor y \rfloor \wedge x \in \mathit{Valid}\ s)$$

$$[\![s \vdash m \mapsto m';\ \mathit{Depth}\ (\mathit{Heap}\ s\ m) < \mathit{Depth}\ (\mathit{Heap}\ s\ m')]\!] \Longrightarrow s \vdash m \rightsquigarrow^{T} m'$$

$$\begin{array}{c}[\![s \vdash m \rightsquigarrow^{T} m';\ s \vdash m' \mapsto ma;\ \mathit{Depth}\ (\mathit{Heap}\ s\ m) < \mathit{Depth}\ (\mathit{Heap}\ s\ ma)]\!] \\ \Longrightarrow s \vdash m \rightsquigarrow^{T} ma\end{array}$$

The refinement relation then implies the equivalence of subtrees in the models. We omit the page table and operations here, a complete description of this refinement step will be published in later work.

Further refinement will proceed by independent refinement of the list heap and page table to source level. There are a number of issues to address in this process, including a choice of suitable language for use in the refinement steps once we decompose operations into imperative code.

5 Conclusion

We have presented the initial stage of a refinement process to verify the virtual memory subsystem of the L4 microkernel. We have shown an abstract model of address spaces together with the operations on them that the kernel API offers. We have refined it into a tree-like structure that is conceptually closer to the data structures used in the kernel implementation.

The next step after refining the current stage into a linked list structure and a page table implementation will be source code verification. Even though we have not yet reached the implementation level, the process of building an abstract model and refining it has already had a beneficial impact on the L4 kernel. During the process of developing these models we have encountered and clarified a number of small ambiguities and errors in the reference manual, have identified unnecessary restrictions, and discovered small errors in the implementation.

Our activities in verifying the L4 kernel apart from the memory subsystem include building a complete abstract model of the L4 API that is executable and lends itself to simulation and exploration. We are also looking at how further

safety and security properties like confidentiality and information flow are best formulated in the context of the L4 model we are building.

Acknowledgements We thank Kai Engelhardt, Kevin Elphinstone, Michael Norrish, Adam Wiggins, and the developers on the pistachio-core mailing list for advice and stimulating discussions.

References

1. *L4 eXperimental Kernel Reference Manual Version X.2*, 2004.
2. ARM Limited. *ARM Architecture Reference Manual*, June 2000.
3. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
4. William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., December 1994.
5. Thierry Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.
6. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
7. Gregory Duval and Jacques Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.
8. System Architecture Group. The L4Ka::Pistachio microkernel. White paper, University of Karlsruhe, May 2003.
9. Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.
10. J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.
11. P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, Computer Science Laboratory, SRI International, 1980.
12. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. `http://www.in.tum.de/~nipkow/LNCS2283/`.
13. J. S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, Distributed Systems Laboratory, University of Pennsylvania, 1997.
14. P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.
15. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

XPath Formal Semantics and Beyond: a Coq based approach

Pierre Genevès[1] and Jean-Yves Vion-Dury[1,2]

[1] WAM Project, INRIA Rhône-Alpes
[2] Xerox Research Centre Europe

Abstract. XPath was introduced as the standard language for addressing parts of XML documents, and has been widely adopted by practioners and theoreticaly studied. We aim at building a logical framework for formal study and analysis of XPath and have to face the combinatorial complexity of formal proofs caused by XPath expressive power. We chose the Coq proof assistant and its powerful inductive constructions to rigorously investigate XPath peculiarities. We focus in this paper on a basic modeling of XPath syntax and semantics, and make two contributions. First, we propose a new formal semantics, which is an interpretation of paths as first order logic propositions that turned out to greatly simplify our formal proofs. Second, we formally prove that this new interpretation is equivalent to previously known XPath denotational semantics [20, 18], opening perspectives for more ambitious mathematical characterizations. We illustrate our Coq based model through several examples and we develop a formal proof of a simple yet significant XPath property that compare quite favorably to a former informal proof proposed in [18].

1 Introduction

XML [4] is now becoming the de facto standard for both representing structured documents and exchanging information. This success impacts major parts of the computing infrastructure such as the future world wide web, information systems, and databases. XPath [6] was introduced by the W3C [16] for specifying node selection, matching conditions, and for computing values from an XML document. XPath is part of other XML-related standards such as the transformation language XSLT [5], the modeling language XML Schema [12], the linking standard XLink [8] and the forthcoming XQuery [3] database access language, that is triggering considerable attention from big industrial players. Because of its fundamental role, we see XPath as a cornerstone of XML technologies.

Motivation. We aim at building a rigorous framework for formal study and analysis of XPath. This paper focuses on a basic modeling of XPath data model, syntax and semantics as a first step toward a more ambitious goal, which is to axiomatize and characterize the containment and equivalence relations over XPath expressions. The first problem to address is the combinatorial complexity of proofs caused by XPath structure (e.g. cases analysis, structural inductions). The second problem is to handle incremental variations (and extensions) of the language fragment we want to deal with while maintaining the

established properties. These two difficulties are clearly in favor of using mechanized proofs, but require a proof assistant offering powerful data structure modeling capabilities and providing a specialized language for building complex and modular proof tactics. We chose the Coq proof assistant [7] because of *(i)* its powerful inductive constructions, *(ii)* its type system and *(iii)* its tactics language. Another important point for the authors was the availability of module abstractions (clearly in favor of large project developments) and also of a very good documentation [2] that considerably eased entering Coq's arcanes. Last but not least, Coq is currently a large and active research project offering long term perspectives as well as a good support to a growing user community. Usually, proof assistants allow enforcing and verifying known mathematical results or proving simple but important algorithms. The authors expect from this exploratory work an ambitious step toward offering a common framework to theoreticians and engineers working around XML technologies. We consider XQuery as a potential target since it comes with a very large and complex formal semantics [11] while being probably too complex to support mathematical treatments without the help of a scalable and typed proof assistant (for instance, proving a worthwhile weak type soundness for the query language, or reasoning formally about normalization and optimization).

Contribution. As a first result, we propose a new formal semantics for the XPath language, which is basically an interpretation of XPath expressions in first-order logic. One of the main advantages of this semantics is that both paths and qualifiers get an unified interpretation; thus the general complexity of proofs involving XPath interpretation is greatly reduced. The other expected benefit is to abstract over the usual computational vision and to focus on the intrinsic meaning of the language. Our second contribution is a formal proof of the equivalence of semantics that enables further construction on top of this simple logical interpretation.

Related Work. The first version of the XPath specification [6], published in 1999, describes the meanings of XPath constructs and operators in more than thirty pages of english. A formal semantics of XPath was given in 2000 by Wadler in [20]. This denotational semantics inspired works on theoretical issues around XPath: rewriting [18], query containment [19] and algorithmic complexity [15]. However, this semantics conveys a computational vision and has often been directly translated into poorly efficient functional algorithms [15]. Several authors adopted simpler semantics, focusing on boolean tests or tree patterns [13] thus missing the most innovative and core XPath feature: node-set selection. Recent work on the forthcoming XPath 2.0 language formally defined static and operational semantics [10]. While being able to deal with complex typing issues raised by substantial evolution of the language specification, these semantics are probably too complex for being directly used in useful manual proofs.

Works on XPath containment and equivalence problems identified and conjectured complexity classes for several XPath fragments (see [17] for an overview). However, most of these works rely on manual proofs-by-reduction that do not help for finding sound and complete algorithms on a significant XPath subset. On the opposite, we aim at building a logical and formal framework for studying XPath, and especially for investigating XPath containment in a constructive way.

Outline We first introduce XPath and its data model in section 2. Section 3 presents the basics of XPath semantics: query results, axes and node tests. A denotational semantics of paths inspired from established contributions is then described in section 4, which also highlights its drawbacks for formal proofs. Section 5 introduces our new logical semantics and illustrates the interest of its Coq modeling through the demonstration of an XPath property. Before concluding, section 6 summarizes the formal proof of the equivalence of both semantics, constructed using the Coq proof system.

2 XPath Syntax and Data Model

A tree document model. XPath considers an XML document as a tree with several kinds of nodes (root, element, text, attribute, namespace, processing instruction, and comment). The tree is built by a successful parsing of a well-formed XML document. The tree contains only one root node, which has no parent, no attribute and no namespace node, but that may have any other kind of nodes as children. Only elements can have children. Nodes are fully connected using the relation $\rightarrowtriangle$ that maps a node to its children, and the reflexive and transitive closure $\rightarrowtriangle^*$ of this relation. Moreover, a total ordering relation $\ll$ between any two elements reflects the depth-first traversal order of the tree. We implemented this document model in Coq as two separate modules "XNodes" and "XTree" that respectively define the types "Node" and "Tree" which we refer to in this paper.

XPath expressions. In their simplest form XPath expressions look like "directory navigation paths". For example, the XPath expression

/book/chapter/section

navigates from the root of a document (designated by the leading slash "/") through the top-level "book" element to its "chapter" child elements and on to its "section" child elements. The result of the evaluation of the entire expression is the set of all the "section" elements that can be reached in this manner, returned in the order they occurred in the document. At each step in the navigation the selected nodes for that step can be filtered using qualifiers. A qualifier is a boolean expression between brackets that can test path existence. So if we ask for

/book/chapter/section[citation]

then the result is *all* "section" elements that have at least one child element named "citation". The situation becomes more interesting when combined with XPath's capability of searching along "axes" other than the shown "children of" axis. Indeed the above XPath is a shorthand for

/child::book/child::chapter/child::section[child::citation]

where it is made explicit that each *path step* is meant to search the "child" axis containing all children of the previous context node. If we instead asked for

/child::book/descendant::*[child::citation]

then the last step selects nodes of any kind that are among the descendants of the top element "book" and have a "citation" child element. Previous examples are all *absolute* XPath expressions (since they involve a leading "/"). The general meaning of an expression is defined relatively to a context node in the tree. Starting from a particular context node in the tree, every other nodes can be reached. This is because XPath defines powerful navigational capabilities, including a full set of axes, as captured on figure 1. For more informal details on the complete XPath language, the reader can refer to the specification [6].

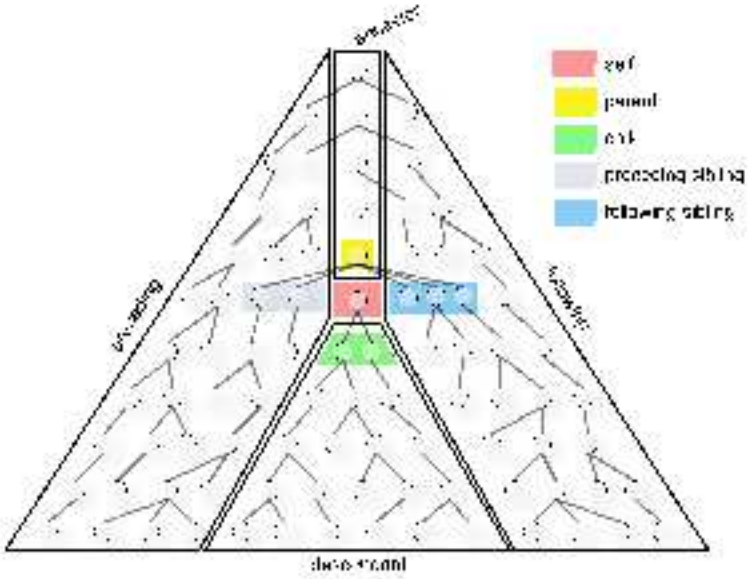


Fig. 1. Axes: partitions of document nodes from a particular context node.

Abstract syntax: a compositional fragment. For the remaining part of the paper, we focus on a restricted but significant fragment of XPath, composed of all XPath axes. The abstract syntax of the fragment is given on figure 2. In order to make the XPath syntax fully compositional, two variants are included: the void path ϵ and the explicit root node ϵ (respectively proposed in [18] and [19]). An other extension concerning qualifiers is the inclusion constraint $p_1 = p_2$ over set of nodes selected by p_1 and p_2. First defined in [19], the authors believe that this feature brings useful expressive power without increasing cost of formal treatment (however this will be verified along our ongoing work on path containment). Note that it turns the construct $p_1[p_2]$ into a syntactic sugar for p_1[**not** $(p_2 \mid \epsilon)$]. Although the XPath fragment we consider already covers a significant range of real world use cases, our intent is to extend it to cover the XPath standard as much as possible.

$$
\begin{array}{lll}
\textit{Path} & p & ::= p/p \mid p[q] \mid p \triangleleft p \mid p \cap p \mid (p) \mid a{::}N \mid \bot \mid \top \\
\textit{Qualifier} & q & ::= q \textbf{ and } q \mid q \textbf{ or } q \mid \textbf{not } q \mid p \mid p \sqsubseteq p \mid \textbf{true} \mid \textbf{false} \\
\textit{Axis} & a & ::= \text{child} \mid \text{descendant} \mid \text{self} \mid \text{descendant-or-self} \\
 & & \quad \mid \text{following-sibling} \mid \text{following} \mid \text{parent} \mid \text{ancestor} \\
 & & \quad \mid \text{preceding-sibling} \mid \text{preceding} \mid \text{ancestor-or-self} \\
\textit{NodeTest} & N & ::= n \mid * \mid \textbf{text}() \mid \textbf{comment}() \mid \textbf{element}() \\
 & & \quad \mid \textbf{processing-instruction}() \mid \textbf{node}()
\end{array}
$$

Fig. 2. XPath Abstract Syntax.

Our syntactic modeling in Coq is directly inspired from the abstract syntax. A cross-inductive set definition (see figure 3) models XPath expressions: $\bot$, $\top$, $a{::}N$ are path atoms and **true**, **false** are qualifier atoms, whereas other operators are binary constructors. The definition relies on the definitions of "Axis" and "NodeTest" which are simple set enumerations.

```
Inductive XPath : Set :=
  — void : XPath
  — top : XPath
  — union : XPath → XPath → XPath
  — inter : XPath → XPath → XPath
  — slash : XPath → XPath → XPath
  — qualif : XPath → XQualif → XPath
  — step : Axis → NodeTest → XPath
with XQualif : Set :=
  — not : XQualif → XQualif
  — and : XQualif → XQualif → XQualif
  — or : XQualif → XQualif → XQualif
  — leq : XPath → XPath → XQualif
  — _true : XQualif
  — _false : XQualif.
```

Fig. 3. Set of all XPath expressions in Coq.

Paths inside qualifiers (as p_2 in $p_1[p_2]$) are modeled through a syntaxic sugar:

```
Definition path (p : XPath) : XQualif := not (leq p void).
```

At this stage, XPath expressions can be instanciated using functional notation, for example:

```
slash root (qualif (step child book) (path (step child chapter)))
```

or even with the familiar infix notation:

$$\wedge/\text{book[chapter]}$$

made possible by Coq's notation mechanism and definitions of operators associativity. Although some syntactic properties can already be worked out, involving results of XPath expressions requires further modeling. We formalize and model the interpretation of XPath expressions in the next sections.

3 XPath Semantics: Basics

Result of an expression. The evaluation of an XPath expression returns a node-set: an unordered collection of nodes without duplicates. We chose to model a node-set in Coq as a custom list type (shown on figure 4) rather than a set. This is in order to cope with the "position()" feature in qualifiers [6] and sequences of the forthcoming XPath 2.0 language [1]. Indeed, the "position()" feature requires an ordered representation of selected nodes for filtering purposes. Moreover, XPath 2.0 handles node sequences (ordered collections of zero or more items, with possible duplicates) instead of node-sets. Thus, our Coq modeling of node-sets presently uses a list together with an associated predicate for forcing uniqueness of nodes in the node-set.

```
Inductive NodeSet : Set :=
  — empty : NodeSet
  — item : Node → NodeSet → NodeSet.
```

Fig. 4. Coq modeling of node-sets.

Axes and node tests. The path step $(a::N)$ is the most basic XPath construct that allows to navigate in the tree in order to retrieve a node-set. Its semantics relies on two functions f and $\mathcal{T}$ that respectively define the semantics of an axis a and a node test N. The navigational semantics of axes can be pictured using the tree document model (see figure 1); and more formally defined using the *parent/child* relation (as usual $\rightarrowtriangle^+$ means $\rightarrowtriangle\rightarrowtriangle^*$), and the irreflexive ordering relation $\ll$. The function f retrieves a node-set starting from a context node x:

a	$f(a)_x$
self	$\{x\}$
child	$\{y \mid x \rightarrowtriangle y\}$
parent	$\{y \mid y \rightarrowtriangle x\}$
descendant	$\{y \mid x \rightarrowtriangle^+ y\}$
ancestor	$\{y \mid y \rightarrowtriangle^+ x\}$
descendant-or-self	$\{y \mid x \rightarrowtriangle^* y\}$
ancestor-or-self	$\{y \mid y \rightarrowtriangle^* x\}$
following-sibling	$\{y \mid y \in \text{sibling}(x) \wedge x \ll y\}$
preceding-sibling	$\{y \mid y \in \text{sibling}(x) \wedge y \ll x\}$
preceding	$\{y \mid y \ll x\}$
following	$\{y \mid x \ll y\}$
attribute	$\{y \mid x \rightarrowtriangle y \wedge \text{is-attribute}(y)\}$
namespace	$\{y \mid x \rightarrowtriangle y \wedge \text{is-namespace}(y)\}$
with sibling(x)=	$\{y \mid \exists z \;\; z \rightarrowtriangle x \;\wedge\; z \rightarrowtriangle y\}$

The node test part of a step is useful to filter the nodes according to their kind. The function $\mathcal{T}$ performs the test by attempting to match a node x with the node test N used in the step, according to the table below. The matching depends on the axis used in the step:

N	a	$\mathcal{T}(a, N, x)$
n		name(x)=n
$*$	attribute	is-attribute(x)
$*$	namespace	is-namespace(x)
$*$	other	is-element(x)
text()		is-text(x)
comment()		is-comment(x)
processing-instruction()		is-pi(x)
element()		is-element(x)
node()		true

The functions f and $\mathcal{T}$ are directly translated into Coq definitions that drive our "XTree" document model. The composition of f and $\mathcal{T}$ allows to define the interpretation of a path step, which is an essential aspect of path semantics.

4 Denotational Semantics of Paths and Qualifiers

A classic formal semantics of paths finds its origins in [20], [18] and [19]. A formal semantics function $\mathcal{S}$ computes the node-set selected by a path p starting from a context node x in the tree:

$$
\begin{array}{lcl}
\mathcal{S} : \textit{Path} \longrightarrow \textit{Node} \longrightarrow \text{Set}(\textit{Node}) & & \\
\mathcal{S}[\![\wedge]\!]_x & = & \{x_1 \mid x_1 \twoheadrightarrow^* x \wedge \text{root}(x_1)\} \\
\mathcal{S}[\![\bot]\!]_x & = & \emptyset \\
\mathcal{S}[\![p_1 \mid p_2]\!]_x & = & \mathcal{S}[\![p_1]\!]_x \cup \mathcal{S}[\![p_2]\!]_x \\
\mathcal{S}[\![p_1 \cap p_2]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p_1]\!]_x \wedge x_1 \in \mathcal{S}[\![p_2]\!]_x\} \\
\mathcal{S}[\![p_1/p_2]\!]_x & = & \{x_2 \mid x_1 \in \mathcal{S}[\![p_1]\!]_x \wedge x_2 \in \mathcal{S}[\![p_2]\!]_{x_1}\} \\
\mathcal{S}[\![(p)]\!]_x & = & \mathcal{S}[\![p]\!]_x \\
\mathcal{S}[\![p[q]]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p]\!]_x \wedge \mathcal{Q}[\![q]\!]_{x_1}\} \\
\mathcal{S}[\![a{::}N]\!]_x & = & \{x_1 \mid x_1 \in f(a)_x \wedge \mathcal{T}(a, N, x_1)\}
\end{array}
$$

The interpretation of a qualified path $p[q]$ uses the dual formal semantics function $\mathcal{Q}$ for qualifiers. $\mathcal{Q}$ returns the boolean evaluation of a qualifier q from a context node x:

$$
\begin{array}{lcl}
\mathcal{Q} : \textit{Qualifier} \longrightarrow \textit{Node} \longrightarrow \textit{Boolean} & & \\
\mathcal{Q}[\![\textbf{true}]\!]_x & = & \text{true} \\
\mathcal{Q}[\![\textbf{false}]\!]_x & = & \text{false} \\
\mathcal{Q}[\![q_1 \textbf{ and } q_2]\!]_x & = & \mathcal{Q}[\![q_1]\!]_x \wedge \mathcal{Q}[\![q_2]\!]_x \\
\mathcal{Q}[\![q_1 \textbf{ or } q_2]\!]_x & = & \mathcal{Q}[\![q_1]\!]_x \vee \mathcal{Q}[\![q_2]\!]_x \\
\mathcal{Q}[\![p]\!]_x & = & \mathcal{Q}[\![\textbf{not } (p \sqsubseteq \bot)]\!]_x \\
\mathcal{Q}[\![(q)]\!]_x & = & \mathcal{Q}[\![q]\!]_x \\
\mathcal{Q}[\![\textbf{not } q]\!]_x & = & \neg\mathcal{Q}[\![q]\!]_x \\
\mathcal{Q}[\![p_1 \sqsubseteq p_2]\!]_x & = & \mathcal{S}[\![p_1]\!]_x \subseteq \mathcal{S}[\![p_2]\!]_x
\end{array}
$$

The implementation of $\mathcal{S}$ in Coq requires updatable definitions of common set operations (union, intersection, inclusion) over previously defined node-sets. More interesting are the two XPath-specific constructs p_1/p_2 and $p[q]$ that require an ordered evaluation of subterms. Indeed, the node-set retrieval driven by p_2 and the filter performed by q respectively operate on the results of p_1 and p. This can be captured in Coq via two higher order functions. These functions abstract over the context node used for the evaluation of p_2 and q:

```
Fixpoint product (s : NodeSet) (fs : Node → NodeSet) {struct s} : NodeSet :=
  match s with
  — empty ⇒ empty
  — item a s1 ⇒ union (fs a) (product s1 fs)
  end.

Fixpoint filter (s : NodeSet) (fs : Node → bool) {struct s} : NodeSet :=
  match s with
  — empty ⇒ empty
  — item a s1 ⇒ if fs a then item a (filter s1 fs) else filter s1 fs
  end.
```

The denotational semantics can then be modeled as a fixpoint that returns the node-set selected by a path p from a context node x in a tree t as shown on figure 5.

```
Fixpoint semanS (t : Tree) (p : XPath)
 (x : Node) {struct p} : NodeSet :=
  match p with
  — void ⇒ empty
  — top ⇒ XTree.roots t x
  — slash p1 p2 ⇒ product (semanS t p1 x) (semanS t p2)
  — union p1 p2 ⇒ union (semanS t p1 x) (semanS t p2 x)
  — inter p1 p2 ⇒ inter (semanS t p1 x) (semanS t p2 x)
  — qualif p1 q2 ⇒ filter (semanS t p1 x) (semanQ t q2)
  — step a n ⇒ filter (f a x t) (test_node t n)
  end

 with semanQ (t : Tree) (q : XQualif) (x : Node) {struct q} :
 bool :=
  match q with
  — _true ⇒ true
  — _false ⇒ false
  — not q1 ⇒ if semanQ t q1 x then false else true
  — and q1 q2 ⇒ if semanQ t q1 x then semanQ t q2 x else false
  — or q1 q2 ⇒ if semanQ t q1 x then true else semanQ t q2 x
  — leq p1 p2 ⇒ incl (semanS t p1 x) (semanS t p2 x)
  end.
```

Fig. 5. XPath Denotational Semantics in Coq.

At this stage, XPath interpretation can be used for studying properties involving query results. Consider for example the containment relation, which holds between two XPath expressions p_1 and p_2 when the set of nodes returned by p_1 is included in the set of nodes returned by p_2, for all trees and context nodes. The containment relation can be formally modeled as follows:

```
Variable t:Tree.
Variable x:Node.

 Variable Sle : XPath → XPath → Prop.

 Conjecture Sle_sound: forall (p1 p2 : XPath),
  Sle p1 p2 → incl (semanS t p1 x) (semanS t p2 x)=true.

 Conjecture Sle_complete: forall (p1 p2 : XPath),
  incl (semanS t p1 x) (semanS t p2 x)=true → Sle p1 p2.
```

The general path equivalence relation $\equiv_{\mathcal{S}}$, that holds between two paths that always have the same interpretation, can then be defined:

```
Inductive Sequiv: XPath → XPath → Prop :=
```

— *seq*: *forall* (*p1 p2* : *XPath*), *Sle p1 p2* → *Sle p2 p1* → *Sequiv p1 p2*.

Identifying path equivalence classes is of very first importance for simplifying general formal treatment of XPath. The equivalence relation is particularly crucial for XPath normalization and rewriting issues (see [18] for an application motivated by streaming XML querying). In addition, both equivalence and containment relations are currently of great interest for XML researchers notably because of their implications for integrity constraints checking [9] and database query optimization [14]. Consider the following basic example: if $\forall p$: *XPath*, $p|p \equiv_s p$ holds then $p|p$ can securely be replaced by p for optimization purposes while preserving query semantics. Using the Coq modeling, the proof of $p|p \equiv_s p$ relies on two set-theoretic lemma (idempotence of set union and reflexivity of set inclusion):

Lemma *opt* : *forall* (*p* : *XPath*), *Sequiv* (*union p p*) *p*.
Proof.
intro;*constructor*; *apply Sle_complete*; *simpl*;*rewrite union_idem*;*apply incl_reflexive*.
Qed.

Now consider a more general XPath property, often named "qualifier flattening", that was first given in [18]. This property basically states that nested qualifiers can be seen as paths:

$$\forall p, p_1, p_2 : \textit{Path} \quad p[p_1[p_2]] \equiv_s p[p_1/p_2] \tag{1}$$

This property can be formulated as follows:

Lemma *flatten_qualifs*: *forall* (*p p1 p2*:*XPath*),
Sequiv (*qualif p* (*path* (*qualif p1* (*path p2*)))) (*qualif p* (*path* (*slash p1 p2*))).

The Coq modeling of the denotational semantics allows to prove this property. However, using the denotational semantics in proofs means dealing with combined node-set computation and boolean evaluation. Indeed, the denotational semantics relies on node-set construction for evaluating paths and boolean evaluation for interpreting qualifiers. Subsequently, ad-hoc auxiliary lemma are required for characterizing these two different computational visions, together with their compositional peculiarities. As a consequence, a major drawback is that intrinsic complexity of proofs becomes hidden behind numerous operational considerations. This causes rather long and complex proof terms. Consider for example the proof of (1); it could begin with the following tactic applications:

intros; *constructor*; *apply Sle_complete*.
simpl.

This generates two subgoals that require to deal with mixed node-set construction and boolean evaluation (see appendix A). In the next section, we present a new simple XPath semantics designed to eliminate this computational overload.

5 A Relational Semantics in First-Order Logic

We propose to translate an XPath expression p into a dyadic formula of the first order logic (*FOL*). The semantics function $\mathcal{R}_p$ defines the interpretation of paths in the first order logic. $R_p(x, y)$ holds for all pairs x, y of nodes such that y is accessed from x through the path p:

$$\mathcal{R}_p : \textit{Path} \longrightarrow \textit{Node} \longrightarrow \textit{Node} \longrightarrow \textit{FOL}$$

$$\begin{array}{lcl}
\mathcal{R}_p[\![\wedge]\!]^y_x & = & y \rightarrowtriangle^* x \wedge \text{root}(y) \\
\mathcal{R}_p[\![\bot]\!]^y_x & = & \text{false} \\
\mathcal{R}_p[\![p_1 \circ p_2]\!]^y_x & = & \mathcal{R}_p[\![p_1]\!]^y_x \vee \mathcal{R}_p[\![p_2]\!]^y_x \\
\mathcal{R}_p[\![p_1 \cap p_2]\!]^y_x & = & \mathcal{R}_p[\![p_1]\!]^y_x \wedge \mathcal{R}_p[\![p_2]\!]^y_x \\
\mathcal{R}_p[\![p_1/p_2]\!]^y_x & = & \exists z \quad \mathcal{R}_p[\![p_1]\!]^z_x \wedge \mathcal{R}_p[\![p_2]\!]^y_z \\
\mathcal{R}_p[\![(p)]\!]^y_x & = & \mathcal{R}_p[\![p]\!]^y_x \\
\mathcal{R}_p[\![p[q]]\!]^y_x & = & \mathcal{R}_p[\![p]\!]^y_x \wedge \mathcal{R}_q[\![q]\!]_y \\
\mathcal{R}_p[\![a::N]\!]^y_x & = & y \in f(a)_x \wedge \mathcal{T}(a, N, y)
\end{array}$$

The dual formal semantics function R_q translates qualifiers into monadic formulæ. $R_q(x)$ holds for all nodes x such that the qualifier q is true from the context node x:

$$\mathcal{R}_q : \textit{Qualifier} \longrightarrow \textit{Node} \longrightarrow \textit{FOL}$$

$$\begin{array}{lcl}
\mathcal{R}_q[\![\textbf{true}]\!]_x & = & \text{true} \\
\mathcal{R}_q[\![\textbf{false}]\!]_x & = & \text{false} \\
\mathcal{R}_q[\![q_1 \textbf{ and } q_2]\!]_x & = & \mathcal{R}_q[\![q_1]\!]_x \wedge \mathcal{R}_q[\![q_2]\!]_x \\
\mathcal{R}_q[\![q_1 \textbf{ or } q_2]\!]_x & = & \mathcal{R}_q[\![q_1]\!]_x \vee \mathcal{R}_q[\![q_2]\!]_x \\
\mathcal{R}_q[\![p]\!]_x & = & \mathcal{R}_q[\![\textbf{not } (p \sqsubseteq \bot)]\!]_x \\
\mathcal{R}_q[\![(q)]\!]_x & = & \mathcal{R}_q[\![q]\!]_x \\
\mathcal{R}_q[\![\textbf{not } q]\!]_x & = & \neg\mathcal{R}_q[\![q]\!]_x \\
\mathcal{R}_q[\![p_1 \sqsubseteq p_2]\!]_x & = & \forall z \quad \mathcal{R}_p[\![p_1]\!]^z_x \Rightarrow \mathcal{R}_p[\![p_2]\!]^z_x
\end{array}$$

This semantics abstracts over the usual computation of node-sets. It gives an unified interpretation of paths and qualifiers. This enables further studying and manipulatation of XPath with an exclusive logical vision. The Coq implementation of this semantics, shown on figure 6, basically translates an XPath expression into a logical proposition. Capturing XPath semantics using Coq's basic "Prop" sort greatly reduces the complexity of proof terms. Indeed, dealing with set-handling peculiarities (such as "product" or "filter") is no more required. Proofs involving query results can be accomplished by using built-in Coq's tactics. For example, let us model the containment relation (as "Rle") and the path equivalence relations $\equiv_{\mathcal{R}}$ (as "Requiv") on top of this new logical interpretation:

```
Variable Rle : XPath → XPath → Prop.

Conjecture Rle_sound: forall (p1 p2 : XPath),
  Rle p1 p2 → (forall y:Node, Rp t p1 x y → Rp t p2 x y).
```

```
Fixpoint Rp (t : Tree) (p : XPath) (x y : Node) {struct p} : Prop :=
  match p with
  — void ⇒ False
  — top ⇒ s_in y (XTree.roots t x)=true
  — union p1 p2 ⇒ Rp t p1 x y ∨ Rp t p2 x y
  — inter p1 p2 ⇒ Rp t p1 x y ∧ Rp t p2 x y
  — slash p1 p2 ⇒ exists z : Node, Rp t p1 x z ∧ Rp t p2 z y
  — qualif p q ⇒ Rp t p x y ∧ Rq t q y
  — step a n ⇒ (s_in y (f a x t))=true ∧ (test_node t n y)=true
  end

 with Rq (t : Tree) (q : XQualif) (x : Node) {struct q} : Prop :=
  match q with
  — _true ⇒ True
  — _false ⇒ False
  — not q ⇒ ¬ Rq t q x
  — and q1 q2 ⇒ Rq t q1 x ∧ Rq t q2 x
  — or q1 q2 ⇒ Rq t q1 x ∨ Rq t q2 x
  — leq p1 p2 ⇒ forall z : Node, Rp t p1 x z → Rp t p2 x z
  end.
```

Fig. 6. XPath Logical Semantics in Coq.

```
Conjecture Rle_complete: forall (p1 p2 : XPath),
 (forall y:Node, Rp t p1 x y→ Rp t p2 x y) → Rle p1 p2.

Inductive Requiv: XPath → XPath → Prop :=
 — req: forall (p1 p2 : XPath),    Rle p1 p2 → Rle p2 p1 → Requiv p1 p2.
```

The "flattening qualifiers" property can now be expressed as follows:

$$\forall p, p_1, p_2 : \mathit{Path} \quad p[p_1[p_2]] \equiv_{\mathcal{R}} p[p_1/p_2] \tag{2}$$

As opposed to the lemma (1), the lemma (2) based on $\equiv_{\mathcal{R}}$ can be proved with a few applications of Coq's built-in tactics only:

```
Lemma flatten_qualifs2: forall (p p1 p2:XPath),
Requiv (qualif p (path (qualif p1 (path p2)))) (qualif p (path (slash p1 p2))).
Proof.
intros; constructor; apply Rle_complete; simpl; intros y H; elim H;
intro H0; split; try assumption; intro H2;apply H1; intros z H3; elim H3;
intros H4 H5; elim H5; intros H6 H7; [ elim (H2 H6); exists z — elim (H2 H4)];
split; try assumption; intro H8;apply (H8 z);assumption.
Qed.
```

The reader will notice that the proof of (2) is even comparable in size with the *manual* proof of (1), found in [18], that expands the denotational semantics:

$$\begin{aligned}
\mathcal{S}[\![p[p_1[p_2]]]\!]_x &= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge \mathcal{Q}[\![p_1[p_2]]\!]_{x_1}\} \\
&= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\mathcal{S}[\![p_1[p_2]]\!]_{x_1} \neq \emptyset)\} \\
&= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\{x_2 | x_2 \in \mathcal{S}[\![p_1]\!]_{x_1} \wedge (\mathcal{S}[\![p_2]\!]_{x_2} \neq \emptyset)\} \neq \emptyset)\} \\
&= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\{x_2 | x_2 \in \mathcal{S}[\![p_1]\!]_{x_1} \wedge x_3 \in \mathcal{S}[\![p_2]\!]_{x_2}\} \neq \emptyset)\} \\
&= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\mathcal{S}[\![p_1/p_2]\!]_{x_1} \neq \emptyset)\} \\
&= \mathcal{S}[\![p[p_1/p_2]]\!]_x.
\end{aligned}$$

To summarize, the Coq proof system and our modeling of XPath offer the major advantages we are interested in:

- rigour of a mechanized inference system in a precisely defined logic framework;
- ability to tackle combinatorial issues by using tactic composition;
- ability to achieve "incremental proving" thanks to proof replaying and updating facilities.

Incremental proving is convenient since it allows to handle the XPath language progressively and to update the semantics accordingly. Last but not least, all these advantages come at a low cost when using our logical semantics, which greatly simplifies proof development.

6 Equivalence of Denotational and Logical Semantics

To ensure that the formal semantics function R_p really captures XPath semantics, we built a formal proof with Coq that shows that denotational and logical semantics are equivalent:

Proposition 1. *Equivalence of semantics.* $\forall p$*: Path*, $\forall x, y$*: Node*, $y \in \mathcal{S}[\![p]\!]_x \;\Leftrightarrow\; \mathcal{R}_p[\![p]\!]_x^y$

The proof uses the modelings presented in sections 4 and 5. Proposition 1 is formulated as follows:

Theorem *sem_equivalence*:
forall (*p* : *XPath*) (*x y* : *Node*) , *s_in y* (*semanS t p x*)=*true* ↔ *Rp t p x y*.

Where "s_in" simply tests the membership of a node in a given node-set. Since paths are inductively defined, the proof naturally uses an induction on p. However, because the definition of paths is cross-inductive with the definition of qualifiers (see figure 3), a mutual induction scheme is used. It is required to prove property 1 for the inductive case $p[q]$, otherwise not possible without assuming the dual property for qualifiers. The appropriate mutual induction scheme (*XJ1*) can be automatically built by Coq from the definition of paths:

Scheme *XJ1* := *Induction for XPath Sort Prop*
with XJ2 := *Induction for XQualif Sort Prop*.

The dual property for qualifiers is defined:

```
Definition sem_equivalence_for_qualifs (q : XQualif) : Prop :=
  forall x : Node, (semanQ env t q x)=true ↔ Rq t q x.
```

The proof of proposition 1, whose skeleton is shown on figure 7, can then begin by applying the mutual induction scheme on p. We attempted to build the proof in a modular way, so that when XPath constructs are changed or added, proof parts of unchanged constructs remain valid. To this end, several tactics named "Solve_X" are defined with the intent to deal with a particular subgoal of the proof. The main proof body (see figure 7) consists in composing these tactics. Each tactic is applied in a way that either completely solve a subgoal or does not modify it at all. This allows to control which parts of the proof require an update when the underlying definitions evolve. Each tactic first attempts to match the goal it is intended to solve and the corresponding

```
Theorem sem_equivalence:
forall (p : XPath) (x y : Node) , s_in y (semanS t p x)=true ↔ Rp t p x y.
Proof.
intro p.
pattern p in ⊢ ×.
apply XJ1 with sem_equivalence_for_qualifs; intros; split;intros;
   try solve_void1;try solve_void2;
   try solve_top1; try solve_top2;
   try solve_union1; try solve_union2;
   try solve_inter1; try solve_inter2;
   try solve_product1; try solve_product2;
   try solve_qualif1; try solve_qualif2;
   try solve_step1; try solve_step2;
   try solve_not1; try solve_not2;
   try solve_and1; try solve_and2;
   try solve_or1; try solve_or2;
   try solve_leq1; try solve_leq2;
   try solve [simpl;auto];
   try solve [simpl;reflexivity];
   try solve [simpl in H;auto;discriminate];
   try solve [simpl in H;auto].
Qed.
```

Fig. 7. Main body of the modular proof of semantics equivalence.

hypotheses. For example, the tactic named "Solve_product1" (see figure 8) isolates the proof of the first inductive case for the "product" construct, whereas the tactic named "Solve_product2" contains the proof of the reciprocal property. In each tactic, the variable names used for matching purposes (e.g. strings after the "?") in the proof context directly correspond to the names that Coq would generate if the proof is manually achieved step by step. Preserving compatibility of names is convenient for updating

```
Ltac solve_product1:=
  match goal with
  — H1: s_in ?y (semanS ?t (slash ?x ?x0) ?x1) = true,
    H: (forall (gx0 gy : Node)(gt : Tree),
        ((s_in gy (semanS gt ?x gx0) = true) ↔ Rp gt ?x gx0 gy)),
    H0:(forall (hx hy : Node)(ht : Tree),
        ((s_in hy (semanS ht ?x0 hx) = true) ↔ Rp ht ?x0 hx hy))
    ⊢ Rp ?t (slash ?x ?x0) ?x1 ?y
  ⇒ simpl in ⊢ ×; simpl in H1;
assert (H2 := in_product1 y (semanS t x x1) (semanS t x0) H1);
elim H2;intros x2 H3; elim H3; intros H3A H3B;exists x2;
elim (H x1 x2 t); intros HE1 HE2;
elim (H0 x2 y t); intros HF1 HF2;
split; [ apply HE1;assumption —apply HF1;assumption]
end.
```

Fig. 8. A tactic for solving a specific subgoal.

proofs, as the proof script can simply be copied and pasted to and from the proof engine. Tactics can use auxiliary lemma that characterize peculiarities of the denotational

```
Lemma in_product1: forall (y : Node)(s : NodeSet)(f:Node→NodeSet),
s_in y (product s f) = true → exists z : Node, s_in z s=true ∧ s_in y (f z) = true.
Proof.
induction s;
[ intros; rewrite product_empty in H; rewrite in_sem1 in H; discriminate
— intros;simpl;cut ({s_in y (product s f) = true} + {s_in y (product s f) = false});
  [ intros HC; elim HC; intros HC1;
    [ elim (IHs f); intros;
      [ exists x; elim H0; intros; split;
        [ apply in_sem5; assumption — assumption] — assumption]
    — exists a; split; [ apply in_sem2
                        — eapply in_Lunion;[ apply H; assumption — assumption]]]
  —apply in_dec]].
Qed.
```

Fig. 9. Lemma for characterizing a peculiarity of the denotational semantics.

semantics. For example, the lemma "in_product1", shown on figure 9 is used by the tactic "Solve_product1" (figure 8). "in_product1" basically states that when the result of a path construct p_1/p_2 is not empty then at least one result node of p_1 is used for evaluating p_2. This is proved using several trivial lemmas on node-sets pictured on figure 10. Proposition 1 allows to securely take advantage of the logical semantics.

Lemma *product_empty* : *forall f* : *Node* → *NodeSet, product empty f = empty*.
Lemma *in_sem1* : *forall a* : *Node, s_in a empty = false*.
Lemma *in_sem2* : *forall* (*a* : *Node*) (*s* : *NodeSet*), *s_in a* (*item a s*) = *true*.
Lemma *in_sem5* : *forall* (*a b* : *Node*) (*s* : *NodeSet*), *s_in a s = true* → *s_in a* (*item b s*) = *true*.
Lemma *in_Lunion* : *forall* (*a* : *Node*) (*s1 s2* : *NodeSet*),
s_in a (*union s1 s2*) = *true* → *s_in a s2 = false* → *s_in a s1 = true*.
Lemma *in_dec* : *forall* (*s* : *NodeSet*) (*a* : *Node*), {*s_in a s = true*} + {*s_in a s = false*}.

Fig. 10. Trivial lemma on node-sets used by proof of "in_product1".

7 Conclusion

In this paper, we focused on a basic modeling of XPath syntax and formal semantics for using the Coq proof system. We introduced a new formal semantics for XPath, that has two main advantages: first, it unifies path and qualifier interpretations. Second, it allows to focus on the intrinsic meaning of XPath from a pure logic point of view. These advantages allow significant simplifications in formal proofs. In addition, we formally proved that this new interpretation is equivalent to the previously known XPath semantics.

Lessons learned. Modeling XPath within the Coq proof system has shown to be a good choice for building a scalable logical framework around XPath. Indeed, Coq's tactic composition features are a realistic way to cope with combinatorial issues raised by XPath expressions. Moreover, Coq provides facilities for incrementally updating proofs when our XPath fragment evolves.

Future Directions We plan to take part of this framework for studying longer and more complex proofs around XPath open questions. Especially, our intent is to axiomatize the containment relation over XPath expressions; and then to demonstrate the soundness and possibly the completeness of the relation. This characterization will strongly rely on the Coq modeling of our logical semantics. After defining the relation, we plan to demonstrate the properties "Rle_sound" and "Rle_complete" presented as conjectures in section 5. The next step is to progressively extend the XPath fragment to support significant real world applications.

References

1. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, W3C working draft, August 2003. http://www.w3.org/TR/2003/WD-xpath20-20030822.
2. Y. Bertot and P. Castéran. *Coq'Art*, chapter To appear. Springer-Verlag, 2004.
3. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C working draft, November 2003. http://www.w3.org/TR/xquery/.
4. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (third edition), W3C recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

5. J. Clark. XSL transformations (XSLT) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.
6. J. Clark and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.
7. The coq proof assistant, 2003. http://coq.inria.fr.
8. S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) version 1.0, W3C Recommendation, June 2001. http://www.w3.org/TR/xlink/.
9. A. Deutsch and V. Tannen. Containment and integrity constraints for xpath fragments. In *Knowledge Representation Meets Databases*, 2001.
10. D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft, February 2004. http://www.w3.org/TR/xquery-semantics/.
11. D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. Xquery 1.0 and xpath 2.0 formal semantics, February 2004. http://www.w3.org/TR/xquery-semantics/.
12. D. C. Fallside. XML Schema part 0: Primer, W3C recommendation, May 2001. http://www.w3.org/TR/xmlschema-0/.
13. S. Flesca, F. Furfaro, and E. Masciari. Minimization of tree patterns queries. In *Proceedings of the 29th VLDB Conf.*, pages 497–508, January 2000.
14. P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *First International Workshop on High Performance XML Processing*, May 2004.
15. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB 2002)*, pages 95–106, Hong Kong, China, 2002. Morgan Kaufmann.
16. MIT, ERCIM, and Keio. The World Wide Web Consortium (W3C), 1994. http://www.w3.org.
17. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory*, pages 315–329. Springer-Verlag, 2002.
18. D. Olteanu, H. Meuss, T. Furche, and F. Bry. Symmetry in XPath. In *Proceedings of Seminar on Rule Markup Techniques, no. 02061, Schloss Dagstuhl, Germany (7th February 2002)*, 2001.
19. J.-Y. Vion-Dury and N. Layaïda. Containment of XPath expressions: an inference and rewriting based approach. In *Extreme Markup Languages*, August 2003.
20. P. Wadler. Two semantics for XPath. http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics/xpath-semantics.pdf, January 2000.

A Denotational interpretations of paths and qualifiers mixed in a proof.

```
2 subgoals

  p : XPath
  p1 : XPath
  p2 : XPath
  ============================
   incl
     (filter (semanS t p x)
        (fun x0 : Node =>
```

```
         if incl
              (filter (semanS t p1 x0)
                 (fun x1 : Node =>
                  if incl (semanS t p2 x1) empty
                  then false
                  else true)) empty
         then false
         else true))
     (filter (semanS t p x)
        (fun x0 : Node =>
         if incl (product (semanS env t p1 x0) (semanS env t p2))
              empty
         then false
         else true)) = true

subgoal 2 is:
 Sle (qualif p (path (slash p1 p2))) (qualif p (path (qualif p1 (path p2))))
```

The Axiomatization of Group Theory: An Experiment in Constructive Set Theory

Xin Yu and Jason Hickey

Department of Computer Science, California Institute of Technology
M/C 256-80, Pasadena, CA 91125, USA

Abstract. We explore a machine-checked formalization of elementary group theory in constructive set theory. Our formalization uses an approach where we start by specifying the group axioms as a collection of inference rules, defining a logic for groups. Then we can derive all properties of groups from these inference rules as well as the axioms of the set theory. The formalization of all other concepts in abstract algebra is based on that of the group. The formalization we present was fully implemented in the MetaPRL theorem prover and all properties of the algebraic objects were formally derived in MetaPRL.

1 Introduction

The notions of abstract algebra are central to many areas of mathematics. Abstract algebra has also made many contributions to computer science, including abstract data types and object-oriented programming. Formalizing abstract algebra in a formal, automated system where proofs can be mechanically generated and verified is valuable: formalization of many areas of mathematics could be based on such abstract algebra theory; and formalization of many computer science concepts could be modeled after it.

Of course, we are far from being the first ones to work with abstract algebra in a formal system. For example, Gunter working with HOL [1] has proved group isomorphism theorems and shown the integers mod n to be an implementation of abstract groups [2]. Jackson has implemented computational abstract algebra in the NuPRL system [3, 4, 5]. And in IMPS [6] there is a notion of *little theories* [7] which they use for proving theorems about groups and rings. Kammüller and Paulson [8] have proved Sylow's theorem in Isabelle-HOL, a large proof that required mechanizing a great deal of group theory.

In this paper, we present a formalization of the abstract algebra concepts in set theory by axiomatization. This is a part of larger effort to explore different approaches to formalizing basic abstract algebra concepts to find out which approach works the best.

Currently most efforts of formalizing algebra using general purpose theorem provers are grounded in type theory. In practice, set theory, as the standard foundation for mathematics, may have an advantage over type theory. Since there is no extensive tradition of presenting mathematics in a type theoretic setting, many techniques for representing mathematical ideas in a set theoretical language

have to be reconsidered for a type theoretical language. In addition, there is much less variation among set theories, in which the well known formulations are defined by a small collection of axioms in the predicate calculus, and for practical purpose, are more or less equivalent [9]. In particular, set theory can often present a convenient framework for developing constructive mathematics using ordinary mathematical concepts.

It is the advantage of set theory over type theory and the fact that abstract algebra is traditionally defined in the language of set theory that motivated us to carry out our implementation of the axiomatization idea for formalizing abstract algebra in a set theory setting. The actual work was done in the constructive set theory of the MetaPRL system [10, 11, 12].

We first specify the group axioms as a collection of inference rules, defining a logic for groups. Then we can tell what it means for a given set together with a binary operation to be a group, and derive all properties of groups from these inference rules as well as the axioms of the set theory. The formalization of other abstract algebra concepts, such as subgroups and homomorphisms, is based on that of the group.

We have proved many theorems of group theory in MetaPRL. As a verification of the method and a good illustration of constructivity, such a machine-checked formalization plays an important role in our implementation. In the interest of space, we only give an overview of our formalization and sketch some proofs in this paper; more details can be found in [12, 13].

Organization. Section 2 introduces our detailed formalization of group theory. Section 3 gives an example of a concrete group, provides a detailed discussion of some properties of our formalization, and suggests some alternative formalization approaches. Section 4 gives conclusions.

1.1 Constructive Set Theory and the CZF module in MetaPRL

Constructive set theory, initiated by John Myhill in 1975 [14], is a theory of sets that, among several others, provides a formal framework for the development of constructive mathematics. It is based on the standard first order language of classical axiomatic set theory and makes no use of constructive notions or objects. Therefore the set theoretical development of constructive mathematics can employ the same ideas, conventions and practice as the set theoretical presentation of classical mathematics. To explain the constructive notion of the set, Aczel introduced Constructive Zermelo-Fraenkel set theory, CZF [15], as a variant of Myhill's constructive set theory and showed its constructiveness by interpreting it in Martin-Löf's type theory [16], which was considered a precise foundation for the constructive approach to mathematics.

Hickey [17] formalized CZF in the MetaPRL logical framework and interactive proof assistant [10,11]. First, he implemented in MetaPRL a constructive Martin-Löf style type theory called ITT (which stands for intuitionistic type theory) similar to NuPRL's one [3]. Next, he derived the axioms of CZF from ITT. Since Aczel's CZF theory is described completely explicitly with a collection of axioms,

after sets and these axioms are encoded in MetaPRL's CZF module, we can use them directly without referring to the type theory.

In CZF, all non-propositional elements of the set theory are sets; the numbers and other structures are coded in the usual manner. Sets use an *extensional* equality; two sets are considered equal if they have the same elements. The following concepts have been formalized in MetaPRL's CZF module: extensional set **equality** $s_1 =_s s_2$, **membership** $s_1 \in_s s_2$, first-order **logic** which includes the restricted quantifiers $\forall x \in_s s, P[x]$ and $\exists x \in_s s, P[x]$, and the unrestricted quantifiers $\forall_s x.P[x]$ and $\exists_s x.P[x]$, **subset** $s_1 \subseteq s_2$, **separation** $\{x \in_s s \mid P[x]\}$, **empty set** $\{\}$, **singleton set** $\{s\}$, **binary union** $s_1 \cup s_2$, **general union** $\cup s$, **unordered pairing** (s_1, s_2), and **infinity** (**the natural numbers**) ω. The subscript s in the representations of $s_1 =_s s_2$, etc., means this is set theoretical compared with those type theoretic implementations in MetaPRL's ITT module.

Our formalization of abstract algebra is built on the basis of MetaPRL's CZF implementation.

2 Formalization of Group Theory

2.1 Groups

In mathematics, a group $\langle G, * \rangle$ is defined as a set G together with a binary operation $*$ defined on G that satisfies the following axioms:

G1. $*$ is associative: for any $a, b, c \in G$, $(a * b) * c = a * (b * c)$.
G2. There is a left *identity* element $e \in G$ such that for every $a \in G$, $e * a = a$.
G3. For some left identity element e, there is, for every $a \in G$, at least one left *inverse* element a' such that $a' * a = e$.

A group must satisfy all of the group axioms; and all properties of groups are derived from these axioms. Inspired by this mathematical definition, we use a set theoretic axiomatization to formalize groups in CZF. That is, we first specify the group axioms as a collection of inference rules that any group should satisfy; then all properties of groups are derived from these inference rules as well as the axioms of CZF.

We use term group_g to denote "g is a group" which, theoretically, should be defined as a predicate satisfying axioms G1, G2, and G3:

$$\begin{aligned}\text{group}_g \stackrel{\text{def}}{=} & \forall x, y, z \in g.\text{car}.(x\ g.*\ y)\ g.*\ z =_s x\ g.*\ (y\ g.*\ z) \wedge \\ & \exists e \in_s g.\text{car}.\forall x \in_s g.\text{car}.(e\ g.*\ x =_s x \wedge \exists x' \in_s g.\text{car}.x'\ g.*\ x =_s e),\end{aligned}$$

where g should be an ordered pair $(\text{car}, *)$. In our current implementation though, we consider group_g as an abstract concept with the meaning of "g is a group". The reason for this is that MetaPRL's CZF theory do not yet support ordered pairing. This works fine as far as this paper goes. In the future, however, if we need, for example, functors for groups, then we should unfold this definition of group_g.

In terms of g, we represent the four components of group g, carrier set, binary operation, identity, and inverse operation, with terms car_g, e_g, $*_g$, and $'^g$ respectively[1], which altogether conform to a collection of axioms that are stated as inference rules in the formal system.

G1, G2, and G3 must be included in the collection of axioms since they specify what groups are (see 5-7 in the list below). In addition, in the CZF setting of MetaPRL, some axioms about the well-formedness of the group terms are needed (as number 1 describes). Furthermore, the properties of binary operation, unary operation, etc. are usually taken for granted when working informally on paper; in a mechanized system, they must be stated explicitly, so axioms 2 through 4 are necessary.

1. In the CZF set theory of MetaPRL, anything that is not a proposition should be a set: car_g and e_g are sets; for any sets a and b, $a *_g b$ and a'^g are sets.

$$\frac{\Gamma \vdash \mathrm{group}_g}{\Gamma \vdash \mathrm{car}_g \text{ is a set}}, \qquad \frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set}}{\Gamma \vdash a *_g b \text{ is a set}},$$

$$\frac{\Gamma \vdash \mathrm{group}_g}{\Gamma \vdash e_g \text{ is a set}}, \qquad \frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set}}{\Gamma \vdash a'^g \text{ is a set}}.$$

2. For $*_g$ to be a binary operation on car_g, car_g has to be closed under $*_g$, and *exactly* one element is assigned to each possible ordered pair of elements of car_g under $*_g$, i.e., for any $a, b, c \in \mathrm{car}_g$, if $a = b$, then $a *_g c = b *_g c$ and $c *_g a = c *_g b$.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set} \quad \Gamma \vdash a, b \in_s \mathrm{car}_g}{\Gamma \vdash a *_g b \ \in_s \mathrm{car}_g},$$

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set} \quad \Gamma \vdash c \text{ is a set} \quad \Gamma \vdash a, b, c \in_s \mathrm{car}_g}{\Gamma \vdash a =_s b \Rightarrow a *_g c =_s b *_g c},$$

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set} \quad \Gamma \vdash c \text{ is a set} \quad \Gamma \vdash a, b, c \in_s \mathrm{car}_g}{\Gamma \vdash a =_s b \Rightarrow c *_g a =_s c *_g b}.$$

3. Similarly, for $'^g$ to be a unary operation on car_g, car_g has to be closed under $'^g$ and exactly one element is assigned to each element of car_g under $'^g$.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g}{\Gamma \vdash a'^g \ \in_s \mathrm{car}_g}, \qquad \frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g \quad \Gamma \vdash b \in_s \mathrm{car}_g}{\Gamma \vdash a =_s b \Rightarrow a'^g =_s b'^g}.$$

[1] In MetaPRL, input is in ASCII format, while output is pretty-printed so that it can be easily understood by those unfamiliar with the MetaPRL syntax. For example, we use `car{'g}` for the input of the carrier set of the group in the actual system. In this paper, we try to avoid the ASCII representations and instead use the pretty-printed forms of terms and definitions for clarity.

4. e_g is in car_g.

$$\frac{\Gamma \vdash \mathrm{group}_g}{\Gamma \vdash e_g \in_s \mathrm{car}_g}$$

5. $*_g$ is associative.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set} \quad \Gamma \vdash c \text{ is a set} \quad \Gamma \vdash a, b, c \in_s \mathrm{car}_g}{\Gamma \vdash a *_g (b *_g c) =_s (a *_g b) *_g c}$$

6. e_g is the left identity.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g}{\Gamma \vdash e_g *_g a =_s a}$$

7. $'_g$ is the left inverse operation.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g}{\Gamma \vdash a'^g *_g a =_s e_g}$$

The above inference rules define the axioms for groups. For any instance of a group, we will need to verify the axioms. However, for general groups, many properties are immediate, such as the left inverse/identity is also the right inverse/identity, and $a*b = a*c$ implies $b = c$ given $a, b, c \in G$ for any group $\langle G, * \rangle$. We also proved some theorems that are a little more complicated, such as the uniqueness of the identity and the inverse operation, and the unique solutions for linear equations $a * x = b$ and $y * a = b$ in the group $\langle G, * \rangle$ where $a, b \in G$.

In MetaPRL, these properties are proved in a straightforward way. The basic idea is similar to that done on paper, but since MetaPRL is an interactive system and provides some automated reasoning, some proofs tend to be easier. Meanwhile, since CZF in MetaPRL is not yet sufficiently automated, some extra efforts might be needed in the proofs. For illustration, we present a proof of one of the theorems below.

Suppose we have already proved, from the axioms of groups and CZF, that the left inverse is also the right inverse and now we want to prove the left identity is also the right identity. First we need to add the statement of this theorem to the `Czf_itt_group` module:

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g}{\Gamma \vdash a *_g e_g =_s a}.$$

Our idea for proving it is

$$a *_g e_g =_s a *_g (a'^g *_g a) =_s (a *_g a'^g) *_g a =_s e_g *_g a =_s a,$$

where the second equation holds because of the associativity of $*_g$ and the third holds because the left inverse is also the right inverse.

To prove it in the MetaPRL proof editor, we first need to replace e_g with $a'^g *_g a$, which can be done by a tactic `setSubstT` provided by MetaPRL's CZF

theory. The usage is `setSubstT` $(s_1 =_s s_2)\ i$, which replaces all occurrences of the term s_1 with s_2 in clause i ($i = 0$ implies the conclusion). So we navigate to this rule and apply the `setSubstT` $(e_g =_s a'^g *_g a)$ `0 thenT autoT` tactic.[2]

Two subgoals are generated. The first one,

$$\frac{\Gamma \vdash \text{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \text{car}_g}{\Gamma \vdash e_g =_s a'^g *_g a},$$

is trivial since we have the axiom

$$\frac{\Gamma \vdash \text{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \text{car}_g}{\Gamma \vdash a'^g *_g a =_s e_g}$$

and $=_s$ is symmetric. With the use of the `eqSetSymT` tactic provided by MetaPRL, this subgoal is proved.

As for the second subgoal,

$$\frac{\Gamma \vdash \text{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \text{car}_g}{\Gamma \vdash a *_g (a'^g *_g a) =_s a},$$

we can utilize the associativity axiom G1 by applying the tactic `setSubstT` $(a *_g (a'^g *_g a) =_s (a *_g a'^g) *_g a)$ `0 thenT autoT`, which generates a new subgoal

$$\frac{\Gamma \vdash \text{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \text{car}_g}{\Gamma \vdash (a *_g a'^g) *_g a =_s a},$$

where $a *_g a'^g$ can be replaced with e_g thanks to the right inverse property we have proved. After this substitution, we get the goal of proving $e_g *_g a =_s a$, trivial by the left identity axiom G2. This completes the proof of the theorem.

For a complete list of the theorems we proved, see [12].

2.2 Abelian Groups

With the elementary group concepts formalized, we can go ahead with formalizing the other concepts in group theory, such as the abelian group.

We define the predicate "g is an **abelian** group" as

$$\text{abel}_g \stackrel{\text{def}}{=} \text{group}_g \wedge \forall a, b \in_s \text{car}_g.(a *_g b =_s b *_g a).$$

Since abel_g implies group_g, all the properties of groups hold for abel_g.

[2] The `autoT` tactic performs "automated" proving based on repeated application of several "basic" tactics; and the infix function `thenT` is a tactical used for sequencing: the proof first applies the substitution, and then applies the `autoT` tactic [17].

2.3 Subgroups

A group can have multiple subgroups. For instance, both $\langle \mathbb{Z}, + \rangle$ and $\langle 2\mathbb{Z}, + \rangle$ are subgroups of $\langle \mathbb{Q}, + \rangle$, where $\mathbb{Z}$ is the integer set, $2\mathbb{Z}$ is the set of even integers, and $\mathbb{Q}$ is the set of rational numbers. To specify a subgroup H of a group G, we need at least two parameters, one specifying the group G and another specifying the subgroup H. The predicate "h is a **subgroup** of g" can be defined as

$$\text{subgroup}_{h,g} \stackrel{\text{def}}{=} \text{group}_h \wedge \text{group}_g \wedge \text{car}_h \subseteq \text{car}_g \wedge \forall a, b \in_s \text{car}_h.(a *_h b =_s a *_g b).$$

The last condition ensures that $*_h$ is the induced operation on car_h from car_g.

We proved that if $\text{subgroup}_{h,g}$, then 1) car_h is closed under $*_g$; 2) $e_h =_s e_g$, and $e_g \in_s \text{car}_h$; 3) for all $a \in_s \text{car}_h$, $a'^h =_s a'^g$ and $a'^g \in_s \text{car}_h$.

2.4 The Power Operation

Before formalizing cyclic subgroups and cyclic groups, let us study the "power" operation which is prerequisite for defining cyclic subgroups and cyclic groups.

Suppose $\langle G, * \rangle$ is a group. For any element $a \in G$, we define

$$a^n = \begin{cases} \underbrace{a * a * ... * a}_{n} & \text{if } n > 0 \\ e & \text{if } n = 0 \\ \underbrace{a' * a' * ... * a'}_{-n} & \text{if } n < 0 \end{cases}$$

as the **power operation** of the group $\langle G, * \rangle$ based on a (a is the **base**).

To formalize it, obviously, we need to use mathematical recursion. However, MetaPRL's CZF module does not yet have the integer set or arithmetic on integers defined. Since the MetaPRL definition of CZF is derived from ITT, we can borrow the integers from ITT for use as the recursion variable, and also borrow the mathematical recursion rules from ITT. This is valid since the recursion parameter is n, which means a^n is still a set given a is a set. In other words, under the mathematical recursion of ITT, a^0, $a^1, a^2, \ldots$, and $a^{-1}, a^{-2}, \ldots$ are still sets; all set properties and set operations can be applied to them. By doing this we can also utilize the arithmetic part in the MetaPRL type theory, which is currently much more complete than that in the MetaPRL set theory.

Now let us define the power operation in group g as:

$$(a^n)_g =_s \begin{cases} a *_g (a^{n-1})_g & \text{if } n > 0 \\ e_g & \text{if } n = 0 \\ a'^g *_g (a^{n+1})_g & \text{if } n < 0 \end{cases}$$

where n is of the integer type in ITT and the recursion is also the one in ITT.

From this definition, we can prove, by induction, that the power operation has the following properties:

1. Well-formedness.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash n \in \mathbb{Z}}{\Gamma \vdash (a^n)_g \text{ is a set}}$$

2. The membership is preserved.

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g \quad \Gamma \vdash n \in \mathbb{Z}}{\Gamma \vdash (a^n)_g \in_s \mathrm{car}_g}$$

3. The power operation is functional, which means it computes equal set values for equal base arguments.

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash b \text{ is a set} \\ \Gamma \vdash a \in_s \mathrm{car}_g \quad \Gamma \vdash b \in_s \mathrm{car}_g \quad \Gamma \vdash n \in \mathbb{Z} \quad \Gamma \vdash a =_s b\end{array}}{\Gamma \vdash (a^n)_g =_s (b^n)_g}$$

Also, with the use of arithmetic rules in the ITT type theory, we can prove

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g \quad \Gamma \vdash m \in \mathbb{Z} \quad \Gamma \vdash n \in \mathbb{Z}}{\Gamma \vdash (a^m)_g *_g (a^n)_g =_s (a^{m+n})_g}.$$

2.5 Cyclic Subgroups

The key to formalizing a cyclic subgroup H of group G generated by a is to build the carrier set $H = \{a^n \mid n \in \mathbb{Z}\}$ from a where a^n is the power operation of group G. Since it can also be described as the set of all elements in car_g that are equal to a^n for some $n \in \mathbb{Z}$, we use the separation axiom of CZF to define it as

$$\mathtt{sep}(x \in_s \mathrm{car}_g \mid \exists n \in \mathbb{Z}. x =_s (a^n)_g).$$

Note that we are using a type theoretic existential within the construction; the CZF implementation in MetaPRL allows this.

Now we define "h is a **cyclic subgroup** of g generated by a" as

$$\begin{array}{rl}\mathrm{cyc\_subg}_{h,g,a} \stackrel{\mathrm{def}}{=} & \mathrm{group}_h \wedge \mathrm{group}_g \wedge a \in_s \mathrm{car}_g \wedge \forall a, b \in_s \mathrm{car}_h.(a *_h b =_s a *_g b) \wedge \\ & \mathrm{car}_h =_s \mathrm{sep}(x \in_s \mathrm{car}_g \mid \exists n \in \mathbb{Z}. x =_s (a^n)_g).\end{array}$$

Of course, the cyclic subgroup H of G generated by a is a subgroup of G. It can be easily proved here: since $\mathrm{car}_h =_s \mathrm{sep}(x \in_s \mathrm{car}_g \mid \exists n \in \mathbb{Z}. x =_s (a^n)_g)$, any element in car_h is also in car_g. Thus, car_h is a subset of car_g. All the other requirements for H to be a subgroup of G are satisfied. So, we can conclude $\mathrm{subgroup}_{h,g}$ from $\mathrm{cyc\_subg}_{h,g,a}$.

Equivalently, we can also define $\mathrm{cyc\_subg}_{h,g,a}$ as

$$\mathrm{cyc\_subg}_{h,g,a} \stackrel{\mathrm{def}}{=} \mathrm{subgroup}_{h,g} \wedge a \in_s \mathrm{car}_g \wedge \mathrm{car}_h =_s \mathrm{sep}(x \in_s \mathrm{car}_g \mid \exists n \in \mathbb{Z}. x =_s (a^n)_g).$$

2.6 Cyclic Groups

A group G is cyclic if there exists $a \in G$ such that for every $x \in G$ there is an integer n such that $x = a^n$. We define it as

$$\mathrm{cycg}_g \stackrel{\mathrm{def}}{=} \mathrm{group}_g \wedge \exists a \in_s \mathrm{car}_g.\forall x \in_s \mathrm{car}_g.\exists n \in \mathbb{Z}.x =_s (a^n)_g.$$

The existential quantifiers in the definition are constructive, so given cycg_g, we know what its generator is and each element is to what power of the generator; on the other hand, to conclude cycg_g, we need to find its generator first.

Since a cyclic group must be a cyclic subgroup of itself, when its generator is explicitly known, we can define "g is a cyclic group generated by a" as

$$\mathrm{cycg}_{g,a} \stackrel{\mathrm{def}}{=} \mathrm{cyc\_subg}_{g,g,a},$$

which is equivalent to (by unfolding $\mathrm{cyc\_subg}_{g,g,a}$)

$$\mathrm{cycg}_{g,a} \stackrel{\mathrm{def}}{=} \mathrm{group}\ g \wedge a \in_s \mathrm{car}_g \wedge \mathrm{car}_g =_s \mathrm{sep}(x \in_s \mathrm{car}_g \mid \exists n \in \mathbb{Z}.x =_s (a^n)_g).$$

The last condition might look strange at the first glance. What it actually means is the carrier is such a set that any element in it is to some integer power of a.

We proved that cycg_g is equivalent to $\exists a \in_s \mathrm{car}_g.\mathrm{cycg}_{g,a}$.

A cyclic group must be abelian, which is easy to prove formally. Suppose we want to conclude from cycg_g that abel_g. Since group g is cyclic, it has a generator a and for any two elements x and y of car_g, there exist m and n in $\mathbb{Z}$ such that $x =_s (a^m)_g$ and $y =_s (a^n)_g$. g is abelian requires

$$x *_g y =_s y *_g x, \qquad i.e., \qquad (a^m)_g *_g (a^n)_g =_s (a^n)_g *_g (a^m)_g.$$

We already have the result

$$\frac{\Gamma \vdash \mathrm{group}_g \quad \Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \mathrm{car}_g \quad \Gamma \vdash m \in \mathbb{Z} \quad \Gamma \vdash n \in \mathbb{Z}}{\Gamma \vdash (a^m)_g *_g (a^n)_g =_s (a^{m+n})_g},$$

so it turns out that we need to prove

$$(a^{m+n})_g =_s (a^{n+m})_g,$$

which is trivial by the commutativity of addition on integers.

2.7 Cosets and Normal Subgroups

With the separation axiom, we define the left and right cosets as

$$\mathrm{lcoset}_{h,g,a} \stackrel{\mathrm{def}}{=} \mathrm{sep}(x \in_s \mathrm{car}_g \mid \exists y \in_s \mathrm{car}_h.(x =_s a *_g y)),$$
$$\mathrm{rcoset}_{h,g,a} \stackrel{\mathrm{def}}{=} \mathrm{sep}(x \in_s \mathrm{car}_g \mid \exists y \in_s \mathrm{car}_h.(x =_s y *_g a)).$$

We need to specify the following inference rules for them: an element x is in $\mathrm{lcoset}_{h,g,a}$ if and only if it is in car_g and there exists $y \in_s \mathrm{car}_h$ such that $x =_s a *_g y$

where $\text{subgroup}_{h,g}$ and $a \in_s \text{car}_g$; same with $\text{rcoset}_{h,g,a}$ except that $x =_s y *_g a$. Both the left and right cosets are subsets of car_g.

Then we define the predicate "h is a normal subgroup of g" as

$$\text{normal\_subg}_{h,g} \stackrel{\text{def}}{=} \text{subgroup}_{h,g} \wedge \forall a \in_s \text{car}_g.(\text{lcoset}_{h,g,a} =_s \text{rcoset}_{s,g,a}).$$

We proved that all subgroups of abelian groups are normal.

2.8 Homomorphisms and Isomorphisms

Now let us look at the relationships between groups, which are generally exhibited in terms of a structure-preserving mapping from one group to the other.

For f to be a mapping from H into G, it is required that: 1) $f(a)$ is in G for any a in H; 2) *exactly* one element in G is assigned as $f(a)$ for each a in H.

So, we define "f is a homomorphism from H into G" as

$$\begin{aligned}\text{hom}_{h,g,f} \stackrel{\text{def}}{=} &\text{group } h \wedge \text{group } g \wedge \forall a \in_s \text{car}_h.(f(a) \text{ is a set} \wedge f(a) \in_s \text{car}_g) \wedge \\ &\forall a, b \in_s \text{car}_h.(a =_s b \Rightarrow f(a) =_s f(b)) \wedge \\ &\forall a, b \in_s \text{car}_h.(f(a *_h b) =_s f(a) *_g f(b)).\end{aligned}$$

$\text{hom}_{h,g,f}$ is functional in the sense that for any two equal mappings f and f', $\text{hom}_{h,g,f}$ always implies $\text{hom}_{h,g,f'}$.

To illustrate our formalization of the homomorphism, let us study a simple example—the trivial homomorphism, which is a mapping f_e from a group H into a group G such that $f_e(a) = e_G$ for all $a \in H$. Suppose H and G are represented by h and g respectively. For any $a, b \in_s \text{car}_h$, $f_e(a) =_s f_e(b) =_s e_g$, so $f_e(a)$ is a set, $f_e(a) \in_s \text{car}_g$, and $a =_s b \Rightarrow f_e(a) =_s f_e(b)$. h is a group implies $a *_h b$ is in car_h, so $f_e(a *_h b) =_s e_g$, which in turn is equal to $e_g *_g e_g =_s f_e(a) *_g f_e(b)$. All the conditions for hom_{h,g,f_e} are satisfied; hom_{h,g,f_e} holds.

Homomorphisms preserve group structure. Put differently, if f is a group homomorphism from H into G, we might know the structure of G from that of H. For example, f maps the identity of H to that of G; it also maps the inverse of an element a in H to the inverse of $f[a]$ in G. And if f is *onto* and H is abelian, then G must also be abelian. In addition, if H_1 is a subgroup of H, then the image $f[H_1]$ of H_1 under f is a subgroup of G; if G_1 is a subgroup of G, then the inverse image $f^{-1}[G_1]$ of G_1 is a subgroup of H. We have proved all these properties of homomorphisms in MetaPRL.

Once homomorphism is formalized, the formalization for isomorphism is trivial since an isomorphism is a bijective homomorphism, i.e., it is a homomorphism that is *one to one* and *onto*. We define "$f : H \to G$ is an isomorphism" as

$$\begin{aligned}\text{iso}_{h,g,f} \stackrel{\text{def}}{=} &\text{hom}_{h,g,f} \wedge \forall a, b \in_s \text{car}_h.(f(a) =_s f(b) \Rightarrow a =_s b) \wedge \\ &\forall a \in_s \text{car}_g.\exists b \in_s \text{car}_h.(a =_s f(b)).\end{aligned}$$

2.9 Kernels

Given f is a group homomorphism from H into G, the kernel of f is the subgroup of H whose carrier set is $\{x \in H \mid f(x) = e_G\}$. To describe the homomorphism, three parameters are needed; we also need an extra parameter to specify the kernel itself. We define "k is the kernel of the homomorphism $f : h \to g$" as

$$\text{kernel}_{k,h,g,f} \stackrel{\text{def}}{=} \text{hom}_{h,g,f} \wedge \text{subgroup}_{k,h} \wedge \text{car}_k =_s \text{sep}(x \in_s \text{car}_h \mid f(x) =_s e_g).$$

Noticing that

$$\text{subgroup}_{k,h} \stackrel{\text{def}}{=} \text{group}_k \wedge \text{group}_h \wedge \text{car}_k \subseteq \text{car}_h \wedge \forall a, b \in_s \text{car}_k.(a *_k b =_s a *_h b),$$

where group_h is implied in $\text{hom}_{h,g,f}$, and $\text{car}_k \subseteq \text{car}_h$ is implied in $\text{car}_k =_s \text{sep}(x \in_s \text{car}_h \mid f(x) =_s e_g)$, we can update the kernel formalization to be

$$\begin{aligned}\text{kernel}_{k,h,g,f} \stackrel{\text{def}}{=} \text{hom}_{h,g,f} \wedge \text{group}_k \wedge \text{car}_k =_s \text{sep}(x \in_s \text{car}_h \mid f(x) =_s e_g) \wedge \\ \forall a, b \in_s \text{car}_k.(a *_k b =_s a *_h b).\end{aligned}$$

This definition implies that if $\text{kernel}_{k,h,g,f}$ then $\text{subgroup}_{k,h}$.

3 Discussions

3.1 The Formalization of a Specific Group

We have successfully formalized most of the fundamental concepts in group theory. Now the question is: under this formalization, given a set, a binary operation, an identity, and an inverse operation, how can we know whether they form a group or not?

Recall the definition of a group. A group must satisfy all those axioms. So first we define car_h, $*_h$, e_h, $'_h$ as the given set, binary operation, identity, and inverse operation respectively. Then without making the assumption group_h, check whether all the axioms of groups (number 1-7 in Section 2.1) are satisfied. If not, we can conclude this composition is not a group at all. If yes, we conclude they do form a group and thus all the proven group properties apply to it. The negative case is easy to understand. For the positive case, let us examine a concrete example, the Klein 4-group, to illustrate this method.

	e	a	b	c
e	e	a	b	c
a	a	e	c	b
b	b	c	e	a
c	c	b	a	e

Fig. 1. Group table of the Klein 4-group

The **Klein 4-group** contains four elements, its group table listed in Fig. 1.

Let us call the Klein 4-group $klein_4$ and declare k_0, k_1, k_2, k_3 as its four elements. Its carrier set, binary operation, identity, and inverse operation can be defined as in Table 1.

With these definitions, we can verify that all of the group axioms are satisfied for $klein_4$, without assuming group_{klein_4}. For example, we can prove the axiom G2 for $klein_4$,

$$\frac{\Gamma \vdash a \text{ is a set} \quad \Gamma \vdash a \in_s \text{car}_{klein_4}}{\Gamma \vdash e_{klein_4} *_{klein_4} a =_s a}.$$

$$\mathrm{car}_{klein_4} \stackrel{\mathrm{def}}{=} \{k_0\} \cup \{k_1\} \cup \{k_2\} \cup \{k_3\}$$
$$e_{klein_4} \stackrel{\mathrm{def}}{=} k_0$$

$k_0 *_{klein_4} k_0 \stackrel{\mathrm{def}}{=} k_0$	$k_1 *_{klein_4} k_0 \stackrel{\mathrm{def}}{=} k_1$	$k_2 *_{klein_4} k_0 \stackrel{\mathrm{def}}{=} k_2$	$k_3 *_{klein_4} k_0 \stackrel{\mathrm{def}}{=} k_3$
$k_0 *_{klein_4} k_1 \stackrel{\mathrm{def}}{=} k_1$	$k_1 *_{klein_4} k_1 \stackrel{\mathrm{def}}{=} k_0$	$k_2 *_{klein_4} k_1 \stackrel{\mathrm{def}}{=} k_3$	$k_3 *_{klein_4} k_1 \stackrel{\mathrm{def}}{=} k_2$
$k_0 *_{klein_4} k_2 \stackrel{\mathrm{def}}{=} k_2$	$k_1 *_{klein_4} k_2 \stackrel{\mathrm{def}}{=} k_3$	$k_2 *_{klein_4} k_2 \stackrel{\mathrm{def}}{=} k_0$	$k_3 *_{klein_4} k_2 \stackrel{\mathrm{def}}{=} k_1$
$k_0 *_{klein_4} k_3 \stackrel{\mathrm{def}}{=} k_3$	$k_1 *_{klein_4} k_3 \stackrel{\mathrm{def}}{=} k_2$	$k_2 *_{klein_4} k_3 \stackrel{\mathrm{def}}{=} k_1$	$k_3 *_{klein_4} k_3 \stackrel{\mathrm{def}}{=} k_0$

$$k_0^{\prime klein_4} \stackrel{\mathrm{def}}{=} k_0 \qquad k_1^{\prime klein_4} \stackrel{\mathrm{def}}{=} k_1 \qquad k_2^{\prime klein_4} \stackrel{\mathrm{def}}{=} k_2 \qquad k_3^{\prime klein_4} \stackrel{\mathrm{def}}{=} k_3$$

Table 1. Definitions for the Klein 4-group

First, since car_{klein_4} is defined as $\{k_0\} \cup \{k_1\} \cup \{k_2\} \cup \{k_3\}$, from the properties of union and singularity, it can be proved that if $a \in_s \mathrm{car}_{klein_4}$, then a must be equal to one of k_0, k_1, k_2, k_3. Then for each of these four cases, by definition,

$$e_{klein_4} *_{klein_4} k_i =_s k_0 *_{klein_4} k_i =_s k_i \quad (i = 0, 1, 2, 3).$$

All the other group axioms can be proved similarly for the $klein_4$ case. Thus we can conclude that this is a group and can make the hypothesis group_{klein_4}. As a consequence, all the group theorems apply for $klein_4$.

The other specific groups can be formalized in the same way.

3.2 Constructivity

Constructivity sometimes makes things harder, especially for work done with machines. For example, classically, there is a theorem "any subgroup of a cyclic group is cyclic." The proving process for the nontrivial case (i.e., the subgroup is other than $\{e\}$ where e is the identity) is assuming G is a cyclic group generated by a and H is a subgroup of G, then supposing m is the smallest integer in $\mathbb{Z}^+$ such that $a^m \in H$, and finally claiming and proving a^m generates H. One of the problems is that in order to assume that m is the smallest natural number such that $a^m \in H$, we need to prove such m exists. In constructive mathematics, the validity of such an existential statement would imply being able to actually compute m. In a straightforward formulation like the one we have implemented, this is not generally possible (since the group membership could be undecidable).

On the other side, constructivity sometimes has advantages. For example, we can extract computational content from the proofs, which allows us to use our formalism for developing guaranteed correct formal abstract algebra algorithms by extracting them from proofs of existentials. However, algorithms extracted naively from proofs are often inefficient, as is the case for MetaPRL for now. Although Caldwell [18] and Nogin [19] demonstrate methods to address this problem, we have not explored this option in detail in MetaPRL.

3.3 Limitations and Alternatives of the Formalization

As discussed above, our formalization of the foundations of abstract algebra — mainly the group theory — is a success: All the major group concepts are formalized; whether a set-operation combination is a group or not can be decided; most theorems and properties can be proved effectively.

Compared with type theory, set theory is more natural in some cases in formalizing algebra. For example, types use intensional equality, but we often care more about extensional properties of algebraic objects.

However, our formalization still has some limitations. For now, it is impossible to quantify over groups and to have sets of groups. But this can be easily fixed if we expand the definition of group_g as mentioned in Section 2.1, that is, we define a group as being an ordered pair of a carrier and a binary operation with axioms specifying the associativity of the operator, the existence of an identity element, and an existence of an inverse for each group element. Anther benefit of doing this is that we need no more to explicitly give names for the identity and the inverse operation. Besides, if we add universal levels to the CZF set theory, then we can also describe the category of all groups.

We tried to limit ourselves to pure CZF, although we still ended up using a few elements of type theory when some parts of MetaPRL's CZF theory were not yet implemented. It could be beneficial to try to clean that up and come up with a truly pure-CZF implementation. On the other hand, we may want to try to take advantage of the availability of the embedding of CZF into ITT in MetaPRL by allowing ourselves to use the type theoretic concepts more freely in our formalization. This way we might be able to come up with some natural "hybrid" formalization where some aspects are formalized using set theoretic concepts and some using type theoretic concepts, picking the most natural approach in every case.

In addition, the formalization is somewhat awkward because typing axioms are not cleanly separated from the principal algebra axioms. We proposed another formalization method of abstract algebra in MetaPRL's ITT theory, which is based on the use of the dependent record type, and in which all objects are first-class and the type information is cleanly separated [20].

4 Conclusions

This paper presents a formal, mechanically verifiable account of foundations of abstract algebra in set theory. We use set axiomatization to formalize groups. Every group should agree with all of the group axioms and all properties of groups are derived from the group axioms and set axioms. We further formalize subgroups, cyclic groups, homomorphisms, and other concepts in group theory on the basis of the formalization of groups. Rings, fields and more advanced abstract algebra can be formalized in constructive set theory based on the group formalization.

Although our work is still elementary and has some limitations, overall the idea is natural (easy to understand), the formalization is easy to use (both for

proving purposes and for extending purposes), and the limitations are more due to the incompleteness of our CZF implementation in MetaPRL than due to the inefficiency of this formalization method or the fault of the CZF theory itself. We believe it will have wide applications in the future.

5 Acknowledgments

The authors would like to thank Aleksey Nogin and the anonymous referees whose valuable observations have greatly improved the contents and the presentation of the paper. We also want to thank Alexei Kopylov for discussions on the formalization.

References

1. Gordon, M., Melham, T.: Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
2. Gunter, E.: Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Logic & Computation 09, Department of Computer and Information Science, Moore School of Engineering, University of Pennsylvania (1989) Distributed with the HOL system in the directory Training/studies/intmod/doingalgpaper.
3. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the NuPRL Proof Development System. Prentice-Hall, NJ (1986)
4. Jackson, P.B.: Exploring abstract algebra in constructive type theory. In Bundy, A., ed.: Automated Deduction – CADE-12; Proceedings of the 12th International Conference on Automated Deduction. Volume 814 of Lecture Notes in Artificial Intelligence., Springer-Verlag (1994) 590–604
5. Jackson, P.B.: Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra. PhD thesis, Cornell University, Ithaca, NY (1995)
6. Farmer, W.M., Guttman, J.D., Thayer, F.J.: IMPS: An interactive mathematical proof system. Journal of Automated Reasoning **11** (1993) 213–248
7. Farmer, W.M., Joshua D. Guttman, F.J.T.: Little theories. In Kapur, D., ed.: Automated-Deduction-CADE-11. Lecture Notes in Artificial Intelligence, New York, Springer-Verlag (1992) 567–581
8. Kammüller, F., Paulson, L.C.: A formal proof of Sylow's first theorem – an experiment in abstract algebra with Isabelle HOL. Journal of Automated Reasoning **23** (1999) 235–264
9. Gordon, M.J.C.: Merging HOL with set theory: preliminary experiments. Technical Report 353, University of Cambridge Computer Laboratory (1994)
10. Hickey, J.J., Nogin, A., Kopylov, A., et al.: (MetaPRL home page) `http://metaprl.org/`.
11. Hickey, J., Nogin, A., Constable, R.L., Aydemir, B.E., Barzilay, E., Bryukhov, Y., Eaton, R., Granicz, A., Kopylov, A., Kreitz, C., Krupski, V.N., Lorigo, L., Schmitt, S., Witty, C., Yu, X.: MetaPRL — A modular logical environment. In Basin, D., Wolff, B., eds.: Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Volume 2758 of Lecture Notes in Computer Science., Springer-Verlag (2003) 287–303

12. Hickey, J.J., Aydemir, B., Bryukhov, Y., Kopylov, A., Nogin, A., Yu, X.: (A listing of MetaPRL theories) http://metaprl.org/theories.pdf.
13. Yu, X.: Formalizing abstract algebra in constructive set theory. Master's thesis, California Institute of Technology (2002)
14. Myhill, J.: Constructive set theory. Journal of Symbolic Logic **40** (1975) 347–382
15. Aczel, P., Rathjen, M.: Notes on constructive set theory. Technical Report 40, Mittag-Leffler (2000/2001)
16. Martin-Löf, P.: Intuitionistic Type Theory. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli (1984)
17. Hickey, J.J.: The MetaPRL Logical Programming Environment. PhD thesis, Cornell University, Ithaca, NY (2001)
18. Caldwell, J.: Moving proofs-as-programs into practice. In: Proceedings of the 12th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (1997)
19. Nogin, A.: Writing constructive proofs yielding efficient extracted programs. In Galmiche, D., ed.: Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics. Volume 37 of Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (2000) http://www.elsevier.nl/gej-ng/31/29/23/67/22/show/Products/notes/index.htt#005.
20. Yu, X., Nogin, A., Kopylov, A., Hickey, J.: Formalizing abstract algebra in type theory with dependent records. In Basin, D., Wolff, B., eds.: 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Emerging Trends Proceedings, Universität Freiburg (2003) 13–27 http://nogin.org/papers/formalaa.html.