TR-CS-98-08

Theorem Proving in Higher Order Logics: Emerging Trends

Proceedings of the 11th International Conference, TPHOLs’98
Canberra, Australia, September–October 1998

Supplementary Proceedings

Jim Grundy and Malcolm Newey
(editors)

September 1998

Joint Computer Science Technical Report Series
Department of Computer Science
Faculty of Engineering and Information Technology
Computer Sciences Laboratory
Research School of Information Sciences and Engineering
This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

Technical.Reports@cs.anu.edu.au

A list of technical reports, including some abstracts and copies of some full reports may be found at:

http://cs.anu.edu.au/techreports/

Recent reports in this series:


Preface

Mechanical theorem provers for higher order logics have been successfully applied in many areas including hardware verification and synthesis; verification of security and communications protocols; software verification, transformation and refinement; compiler construction; and concurrency. The higher order logics used to reason about these problems and the underlying theorem prover technology that support them are also active areas of research. The International Conference on Theorem Proving in Higher Order Logics (TPHOLs) brings together people working in these and related areas for the discussion and dissemination of new ideas in the field.

TPHOLs’98 continues the conference tradition of having both a completed work and work-in-progress stream. The Papers from the first stream were formally refereed, and published as volume 1479 of LNCS. This, supplementary, proceedings records work accepted under the work-in-progress category, and is intended to document emerging trends in higher-order logic research. Papers in the work-in-progress stream are vetted for relevance and contribution before acceptance. The work-in-progress stream is regarded as an important feature of the conference as it provides a venue for the presentation of ongoing research projects, where researchers invite discussion of preliminary results.

Although the TPHOLs conferences have their genesis in meetings of the users of the HOL theorem proving system, each successive year has seen a higher rate of contribution from the other groups with similar goals, particularly the user communities of Coq, Isabelle, LAMBDA, LEGO, NuPrl, and PVS. Since 1993 the proceedings have been published by Springer as volumes in Lecture Notes in Computer Science series. Bibliographic details of these publications can be found at the back of this book; more history of TPHOLs can be found with further information about the 1998 event at http://cs.anu.edu.au/TPHOLs98/.

The conference was sponsored by the Computer Science Department of The Australian National University (ANU), Intel, the Defence Science and Technology Organisation (DSTO), The Australian Research Council, and ACSys (the Cooperative Research Centre for Advanced Computational Systems). The financial support of these groups is gratefully acknowledged.

Tradition dictates that the organising committee of each TPHOLs conference selects the site of the next conference by conducting a poll of potential attendees. We would like to thank the TPHOLs community for choosing The Australian National University as the conference hosts for 1998. Similarly, we are pleased to announce that in the next TPHOLs will be held in early September 1999 on the French Riviera, where it will be hosted by the Coq and CROAP groups at INRIA Rocquencourt and Sophia-Antipolis.

Jim Grundy and Malcolm Newey
Canberra, September 1998
Conference Organisation

Jim Grundy (ANU)
Malcolm Newey (ANU)

TPHOLs’98 Program Committee

Mark Aagaard (Intel)  Paul Jackson (Edinburgh)
Sten Agerholm (IFAD)  Sara Kalvala (Warwick)
David Basin (Freiburg)  Thomas Kropf (Karlsruhe)
Richard Boulton (Edinburgh)  Tim Leonard (Compaq)
Albert Camilleri (HP)  Paul Loewenstein (Sun)
Tony Cant (DSTO)  Tom Melham (Glasgow)
Robert Constable (Cornell)  Paul Miner (NASA)
Gilles Dowek (INRIA)  Malcolm Newey (ANU)
Amy Felty (Bell Labs)  Sam Owre (SRI)
Mike Gordon (Cambridge)  Christine Paulin-Mohring (LRI)
Jim Grundy (ANU)  Lawrence Paulson (Cambridge)
Elsa Gunter (Bell Labs)  Laurent Théry (INRIA)
Joshua Guttman (Mitre)  Phil Windley (Brigham Young)
John Harrison (Intel)  Wai Wong (Hong Kong Baptist)

Additional Reviewers for this Volume

Brendan Mahony  Mark Staples  Burkhart Wolff
## Contents

**Refereed Papers**

Integrating TPS with ΩMEGA ............................................. 1  
*Christoph Benzmüller and Volker Sorge*

Paul E. Black and Phillip J. Windley ............................... 19  
*Some Theorem Proving Aids*

Verification of the MDG Components Library in HOL ................... 31  
*Paul Curzon, Sofiène Tahar, and Omnane Ait Mohamed*

Simulating Term-Rewriting in LPF and in Display Logic ............... 47  
*Jeremy E. Dawson*

A Prototype Generic Tool Supporting the Embedding of Formal Notations 63  
*Andrew M. Gravell and Chris H. Pratten*

Embedding a Formal Notation: Experiences of Automating the Embedding  
of Z in the Higher Order Logics of PVS and HOL ..................... 73  
*Andrew M. Gravell and Chris H. Pratten*

Building HOL90 Everywhere Easily (Well Almost) ................... 85  
*Elsa L. Gunter*

Program Composition in Coq-UNITY .................................. 95  
*François Marques*

Formally Analysed Dynamic Synthesis of Hardware ................... 105  
*Kong Woei Susanto and Tom Melham*

Requirements for a Simple Proof Checker ............................. 119  
*Geoffrey Watson*

Integrating HOL and RAISE: a practitioner’s approach ............... 127  
*Wai Wong and Karl R. P. H. Leung*

Effective Support for Mutually Recursive Types ........................ 141  
*Peter V. Homeier*

**Author Index** ................................................................... 159
Integrating TPS with $\Omega$MEGA

Christoph Benzmüller and Volker Sorge

Fachbereich Informatik, Universität des Saarlandes
D-66141 Saarbrücken, Germany
{chris,sorge}@cs.uni-sb.de

Abstract. We report on the integration of TPS as an external reasoning component into the mathematical assistant system $\Omega$MEGA. Thereby TPS can be used both as an automatic theorem prover for higher order logic as well as interactively employed from within the $\Omega$MEGA environment. TPS proofs can be directly incorporated into $\Omega$MEGA on a tactic level enabling their visualization and verbalization. Using an example we show how TPS proofs can be inserted into $\Omega$MEGA’s knowledge base by expanding them to calculus level using both $\Omega$MEGA’s tactic mechanism and the first order theorem prover OTTER. Furthermore we demonstrate how the facts from $\Omega$MEGA’s knowledge base can be used to build a TPS library.

1 Introduction

Current theorem provers, whether automatic or interactive ones, usually have strength in some specific domains while lacking reasoning power in others. Therefore there have been several attempts in recent years to combine two or more provers in order to enhance their power and facilities. On one hand these combinations have been done for the purpose of sharing databases between different systems and thereby avoiding the duplication of work by constructing analogous databases for all systems involved [8]. On the other hand there are attempts to integrate several existing provers into a single architecture to make use of various kinds of reasoning strategies and proof procedures in a cooperate system [10]. Furthermore, in some interactive systems, external reasoning components – usually first order automatic theorem provers – are used to support the user when proving a theorem interactively, by automatically justifying simple open subgoals [15].

In this paper we report on an experiment of integrating the higher order theorem proving system TPS [1] into the mathematical assistant $\Omega$MEGA [5] for the benefit of both systems. The integration of higher order reasoning components is highly desirable for $\Omega$MEGA as its database of mathematical theories consists mainly of higher order concepts, so that many problems formulated using this database lie naturally beyond the capabilities of the already integrated first order theorem provers SPASS [20], OTTER [14] and PROTEIN [6]. TPS on the other hand gains a graphical proof display and a component for proof verbalization from its integration with $\Omega$MEGA. Furthermore TPS can in principal be extended
in order to integrate proofs passed from $\Omega$MEGA, such that both systems can be integrated on the same level, perhaps sharing a common knowledge base.

The integration was eased by the fact that TPS and $\Omega$MEGA implement the same logic, i.e. classical higher order logic based on Church’s simply typed \( \lambda \)-calculus. Both systems use variants of Gentzen’s natural deduction calculus (ND) [9] to display proofs and provide all the information, e.g. full type information of terms, that is necessary for syntax translation from one system into the other.

We want to emphasize that the work presented in this paper is essentially an extension of the $\Omega$MEGA system in order to integrate TPS as a powerful external reasoning component.

**Remarks on TPS.** TPS can be used either as fully automatic or as an interactive proof development environment based on an (extended) variant of Gentzen’s natural deduction calculus (ND) [9]. Even in interactive mode the automatic component can be called on subproblems. It uses the mating method (connection method) [3] as reasoning technique and provides several built-in search strategies as well as many options to adjust these strategies interactively or even automatically. Furthermore the automatic prover has the ability of selectively expanding definitions [4] based on a dual instantiation strategy. This strategy provides an effective way to decide which abbreviations to instantiate when searching for a proof. The system also allows the user to interrupt the automatic proof process in order to analyze it and to influence further mating search. A very important feature of TPS for our integration is that each proof found by its automatic component gets automatically transformed into a natural deduction proof based on the work of [16, 17]. Furthermore TPS provides comprehensive library facilities for the maintenance of different kinds of objects, such as problems, (polymorphic) definitions, theorems, rewrite rules or even modes specifying flag-settings connected to previously proven theorems.

**Remarks on $\Omega$MEGA.** The $\Omega$MEGA-system for classical type theory (Church’s typed \( \lambda \)-calculus) [5] is designed as an interactive mathematical assistant system, aimed at supporting proof development in mainstream mathematics. It consists of a variety of tools including a proof planner [12], a graphical user interface LOUI [19], the PROVERB system [11] for translating proofs into natural language and a variety of external systems, such as computer algebra systems [13], constraint solvers and automated theorem provers [15].

The basic calculus underlying $\Omega$MEGA is similar to TPS’, i.e., a variant of ND. However the set of rules in $\Omega$MEGA is smaller then the one in TPS. This stems from the necessity of keeping TPS proofs concise for displaying them in a user-friendly fashion. Therefore certain rules abstract over small subproofs (such as RuleP over proofs in propositional logic, cf. Sec. 3.1). In $\Omega$MEGA however the set of basic ND-rules is just large enough to ensure completeness and all extensions to the basic ND-calculus (e.g. equality substitution) are defined as tactics. Nevertheless, proofs can be both constructed and displayed on several
abstract levels by using a three-dimensional data structure\textsuperscript{1} for representing (partial) proofs. The structure on the one hand enables the user to freely switch back and forth between different abstract levels and on the other hand provides a means for directly integrating results of external reasoners while leaving the expansion to the calculus level to $\Omega$MEGA's tactic mechanism.

We demonstrate the integration of TPS and $\Omega$MEGA with a simple example from set theory, which we will solve for demonstration purposes partly interactive within $\Omega$MEGA while passing two subproblems to TPS.\textsuperscript{2} In order to gain a checkable $\Omega$MEGA calculus level proof, the original TPS proof is inserted on a tactic level and expanded via several levels of abstraction and with the help of the first order theorem prover Otter. While discussing the example we exhibit the advantageous side effects for TPS, (1) that the proof can be directly visualized and (2) its structure displayed graphically in $\Omega$MEGA's user interface LOUI, and (3) with the available different abstraction levels it can be verbalized by the PROVERB system. We show the first two effects with the help of several screenshots and exemplify the latter by giving a short verbalization of a subproof of our theorem. In order to efficiently make use of TPS' ability to selectively expand definitions we describe in Sec. 4 an algorithm that automatically imports definitions and recursively also imports all necessary subconcepts from the $\Omega$MEGA knowledge base to TPS. To explain the working scheme of this algorithm we use another example from set theory.

2 Integrating TPS with $\Omega$MEGA

The integration of TPS into $\Omega$MEGA provides two different modes for its use. One mode is to call TPS as black box system for proving a given subproblem, similar to the use of the first order theorem provers already integrated in $\Omega$MEGA. A second mode offers the possibility of employing TPS as an interactive theorem proving environment itself. While in the first mode the proof search of TPS as a black box can only be influenced by elementary flag settings adjustable as $\Omega$MEGA command parameters, e.g. flags specifying a concrete mating search procedure instead of the standard uniform search strategy, the user may take advantage of all interactive features of TPS when calling it in interactive mode. For both types of integration we currently use a file based communication between the two systems.

2.1 Black Box Integration

We demonstrate the black box integration of TPS into $\Omega$MEGA by using the following example:  

\textsuperscript{1} $\Omega$MEGA's proof datastructure stores the (partial) proof on basic ND-level as well as the more abstract levels containing nodes which are justified with tactics and or methods and which correspond to certain parts of the proof on the underlying level.

\textsuperscript{2} This problem can be solved by TPS even without any interaction.
then \(set2\) is the empty set.” Using \(\\OmegaMEGA\)’s theory naive-set this theorem can be formalized as:

\[
\text{assumption} \quad \neg \exists f, u \in set1 \Rightarrow \exists v, v \in set2 \land (f u) = v
\]
\[
\text{theorem} \quad \forall u \in set1 \Rightarrow \exists v, v \in set2 = \emptyset
\]

where the following polymorphic definitions are provided by the naive-set theory: \(\varepsilon = \lambda x. \lambda m. \ominus m\) and \(\emptyset = \lambda x. \bot\). Even though TPS is able to solve this problem automatically we prove this theorem for demonstration purposes partially interactive with \(\\OmegaMEGA\) and partially automatic with TPS. We begin with introducing the following lemma in \(\\OmegaMEGA\):

\[
\text{lemma} \quad (\neg \exists w, w \in set2) \Rightarrow set2 = \emptyset
\]

Next we apply the implication elimination rule (modus ponens) backwards using the theorem as succedent which splits the original proof problem into two subproblems: (a) showing that \(\neg \exists w, w \in set2\) follows from the assumption \(\neg \exists f, u \in set1 \Rightarrow \exists v, v \in set2 \land (f u) = v\) and (b) showing that the newly introduced lemma is valid. This proof situation is visualized in Fig. 1.

Before applying TPS we need to eliminate in a preliminary step the defined construct \(\varepsilon\) in both the assumption (node 1) and the subgoal (node 5). This is necessary as \(\varepsilon\) is a definition from \(\\OmegaMEGA\)’s knowledge base that is unknown to TPS. We can now call TPS, thereby closing the left branch in the proof tree.

The original proof generated by TPS (see Fig. 3; for detailed information about the occurring TPS-justifications we refer to [2]) can be displayed within \(\\OmegaMEGA\) by applying the command \texttt{show-tps-proof} on node 7. The idea of the indirect proof is to derive a contradiction from \(\exists w, w \in set2\) and the above assumption by showing that there indeed exists an \(f, u\), such that \(\forall u, u \in set1 \Rightarrow \exists v, v \in set2 \land (f u) = v\), namely by choosing \(f\) to be the constant function \(\lambda u. w\).

We now concentrate on the introduced lemma in node 3 and first apply implication introduction rule. This introduces the assumption \(\neg \exists w, w \in set2\) as a hypothesis from which we have to derive that \(set2 = \emptyset\). On this open subgoal we apply the functional extensionality principle (two functions are equal, if they are equal with respect to all of their arguments) introducing \(\forall x. \ominus (set2 x) = (\emptyset x)\) (node 10) as new open node. The tactic equiv\(2\) (see the justification in Fig. 2), which implements the extensionality principle on truth values (equality relation coincides with equivalence relation on truth values), can now be applied backwards to the subterm \(\ominus (set2 x) = (\emptyset x)\) introducing the open proof line \(\forall x. \ominus (set2 x) \equiv (\emptyset x)\). Note that the remaining subproblem, namely to show that

\footnote{In order to be consistent with the screen shots in this paper, we use a notation for both terms and types that is common to \(\\OmegaMEGA\) and TPS. Types are printed as subscripts and we write functional types as \(\alpha \rightarrow \beta\) instead of using the more common notation \(\alpha \rightarrow \beta\). The types \(o\) and \(i\) denote the sets of propositions and individuals. Dots bracket as far to the right as it is consistent with structure of the formula and the logical connectives.}

\footnote{Sets are represented as characteristic functions and \(\bot\) denotes false.}
the latter proof line can be derived from the assumption \( \neg \exists w, w \in \text{set2} \), is a first order problem, provided the definition of \( \emptyset \) is expanded: \( \forall x, (\text{set2} x) \equiv \bot \). Thus we could also close this subproblem by calling a first order prover, but again we use TPS. The complete proof on an abstract proof level is shown in Fig. 2 while the original TPS proof for node 13, that can also be displayed within \( \Omega \text{MEGA} \), is presented in Fig. 4.

Fig. 1. The partial proof tree displayed in reverse order. The original proof goal (node 2) is presented at the top and the assumption (triangle-node 1) at the bottom. The formula contents of all numbered nodes are displayed in the term browser. The status of a node is symbolized via the node’s shape and color, e.g. assumptions and hypotheses are represented as green and magenta triangles, whereas all other nodes are represented as colored circles. Although this print may be black and white, we assume that the reader might be able to notice at least different shades and shapes of the nodes in the presented figures. Red circles (e.g. nodes 3 and 5) denote open subgoals and light or dark blue ones already justified nodes. Whereas the dark blue color indicates that a node is grounded, i.e. justified by a ND-Rule (node 2 and 4), a light blue color indicates that a node is justified by a tactic, which can be further expanded by \( \Omega \text{MEGA} \)'s tactic mechanism in order to obtain a completely ND-Rule based proof. Rhombi indicate coreferences to whole subtrees of a given partial proof in order to omit redundancy in the graphical display.
Fig. 2. The completed proof: Note that the subgoals in node 7 and node 13 are now justified by ΩMEGA’s black box tactic TPS. The light blue color of these nodes indicate that they can be further expanded into proofs on a more detailed level. The parameter in the TPS-justification of node 13 refers to a file containing the proof output generated by TPS.

Fig. 3. The original TPS-proof for node 7

Fig. 4. The original TPS-proof for node 13
2.2 TPS as an Interactive Prover

TPS can also be used as an interactive theorem proving environment connected to \(\Omega\)MEGA. For that TPS can be invoked from within \(\Omega\)MEGA in a separate window and initialized with a specified subproblem. When a proof or a partial proof has been constructed within TPS, the user can send it back to the \(\Omega\)MEGA-system, where the results are integrated as subproof into the overall \(\Omega\)MEGA proof.

As the TPS system provides plenty of very interesting interactive features for a user making it possible to solve many non-trivial problems either fully automatic or with little interaction. The \(\Omega\)MEGA system will gain much from the integration of TPS as an interactive theorem proving environment of its own right.

Note that the power of TPS in proving theorems fully automatically is heavily influenced by availability of a comprehensive library providing useful information such as definitions and proof problems in connection to appropriate modes (lists of flag settings). Thus, a steady enrichment and maintenance of a TPS-library closely connected to \(\Omega\)MEGA’s theory database can steadily increase the power of TPS in proving problems as an automatic tool in \(\Omega\)MEGA. In Sec. 4 we present a way of providing TPS with information from \(\Omega\)MEGA’s knowledge base.

3 A Tactic-Based Proof integration

\(\Omega\)MEGA’s main philosophy is that all integrated systems have to generate enough protocol information, such that either a proof or proof plan, i.e. a proof on an abstract level, can be extracted from this information. Proof plans can then be expanded into a pure and checkable ND-level proof. For instance when calling a first order automated theorem prover, returned proofs are transformed into an intermediate data structure, the refutation graph, which is then translated into a high level \(\Omega\)MEGA-proof plan, consisting of methods, tactics or rules [15]. These proof plans are sound if they can be expanded successfully onto ND-rule level.

The integration of TPS is based on a much simpler proof transformation approach, which becomes possible as TPS itself transforms the automatically generated connections into a TPS-ND-level proof. This means that most but not all justifications refer to a standard ND-rule. There exist for instance justifications such as RuleP or RuleC where the first abbreviates simple derivations in propositional logic, while the latter is a slight modification of the standard exists elimination rule.

3.1 Integration and Expansion

The general idea is to define a new theory TPS in \(\Omega\)MEGA which provides exactly one \(\Omega\)MEGA-tactic for each possible TPS-justification. Using the tactics of this special theory each proof generated by TPS can now be mapped one to one onto an \(\Omega\)MEGA proof. As a consequence each TPS-ND-proof can be visualized in
its original form within $\Omega$MEGA’s graphical user interface LOUI. For instance by expanding the TPS-justified node 7 of Fig. 2 the original TPS-proof can be visualized in $\Omega$MEGA. See Fig. 5 in comparison to the original TPS-proof in Fig. 3. Note that the TPS-proof is now embedded as a subproof into the whole proof. Thus line 2 is no longer a hypothesis but a derived line itself and line 20 is used to justify other proof lines.

Each TPS-tactic of $\Omega$MEGA’s TPS theory contains an expansion information, which maps this TPS-tactic to a derivation built upon the rules and tactics in $\Omega$MEGA’s base theory. The $\Omega$MEGA system allows to execute this transformation interactively line by line or at once for all lines. Furthermore the expansion of a line is reversible, meaning in every stage of the expansion one can get back to the original TPS proof (or even the state before the TPS proof was inserted), by unexpanding nodes.

We now discuss the different types of mappings:

1:1 Mapping. Clearly for many TPS-justifications there exist direct counterparts among $\Omega$MEGA’s ND-rules and tactics. For instance tps*NegIntro is mapped to the ND-rule NotI and tps*Imp-Disj is mapped to the $\Omega$MEGA-tactic Equiv which itself can be further expanded on $\Omega$MEGA’s ND-rule level. Such simple mappings are the standard case as many justifications used in TPS have direct counterparts among $\Omega$MEGA’s ND-rules or tactics.

Mappings with Case Distinctions. As a typical example we consider the TPS justification Neg, which is used as justification for an PushNeg application (pushing an outermost negation symbol inside a term) as well as for an PullNeg application (pulling a negation symbol at outermost position). Thus, depending on the structure of the premise and conclusion line the justification tps*Neg is translated into either one of the $\Omega$MEGA-tactics PushNeg or PullNeg.

Restructuring Mapping. The TPS-rule RuleC is a slight modification of the standard exists elimination rule, which roughly explained does not introduce the concrete instance of the existentially quantified line as a new hypotheses but instead introduces an analogous derived line justified with a special judgment. The point is that in this case, it is necessary to slightly manipulate the proof data structure while mapping RuleC to $\Omega$MEGA’s rule Existe, i.e. the status of the instantiated line and some dependencies between proof lines have to be modified.

External System Mapping. Proof lines in TPS justified with RuleP abbreviate simple derivations in propositional logic which are trivial and would rather worsen the readability of the whole proof. Examples for the usage of RuleP are given in the two subproofs automatically proven by TPS in Figs. 3 and 4.

One approach to translate lines justified by RuleP into an $\Omega$MEGA-proof would be to implement a simple propositional logic proof procedure as a recursive tactic (or better tactical) in $\Omega$MEGA. While this is certainly possible it would
Fig. 5. The original TPS-proof for the first subproblem
contradict one important aspect of OMEGA. The aspect is to make use of other probably more specialized external reasoning systems as soon as this seems to be appropriate and thus to avoid unnecessary reimplementations. Hence instead of implementing the translation we just call another one of the integrated automated theorem provers to expand lines justified by RuleP. As at the present time there is no theorem prover purely specialized on propositional logic integrated in OMEGA. We choose to use OTTER as one of the available first order provers. Therefore the RuleP tactic is mapped as expansion onto a tactic specifying the call of OTTER which in turn produces a subproof when executed.

Figure 6 shows the expansion of RuleP via OTTER into the AndI-rule. In this case of a trivial derivation it seems to be an overkill to call an external reasoner. But firstly not all RuleP applications are that simple. One might consider line 7 in Fig. 4 as a more complicated situation. And on the other hand calls of OTTER are fast enough, even with respect to all necessary translations, that they do not slow down the expansion process considerably.

![Fig. 6. Expansion of the TPS*Rule P tactic](image)

The described transformation approach based on OMEGA’s tactic mechanism allows to integrate arbitrary proofs generated by TPS into OMEGA. The whole proof finally can be expanded on OMEGA’s ground level, i.e. a derivation using only rules assigned to OMEGA’s basic ND-calculus. The grounded proof for our example consists of more than 300 nodes and is shown in Fig. 7. This large number of single lines is due to the fact that OMEGA’s basic ND-calculus is rather small – in fact the idea is to minimize the basic calculus while upholding completeness. For example the basic calculus does not a priori include equality as this is a concept defined via Leibniz-equality, i.e. two functions are equal, iff they share the same properties. Therefore some tactics like equality substitution =subst expand into very large and tedious ground level derivations. Currently OMEGA has been applied to examples containing up to 1000 nodes (proof lines) at ground level, but we have reason to believe that the current system is able to also handle far larger examples.

Furthermore there is currently no advanced cleanup function available, which restructures the proof while eliminating most of the redundant and superfluous nodes in the proof tree. This is due to the fact that the problems arising from such deletions inside the three-dimensional proof data structure and their effects
on the reuse of information from proof constructions in a planning scenario have not yet been completely solved.

3.2 Verbalization

Besides the possibility of graphically displaying proofs, as another feature from its integration with $\Omega$MEGA TPS gains the chance of verbalizing its proofs. The verbalization is done with the help of the PROVERB system [11] which is connected to $\Omega$MEGA and can be called within the graphical user interface. PROVERB was originally developed to translate proofs in first order logic into natural language and is currently extended to cover higher order logic as well. For that reason some of the verbalized higher order logic proofs still lack conciseness.

We demonstrate the use of PROVERB by giving the automatically generated natural language proof for the subproblem shown in Fig. 3.

Assumptions:
(1) $\exists f. \forall u. set_1(u) \Rightarrow \exists v. set_2(v)$ and $f(u) = v$.

Theorem: $\exists x. set_2(x)$. Proof: Let $\exists x. set_2(x)$. Let $x'$ be such $x$.

$\forall f. \exists u. set_1(u) \Rightarrow \exists v. set_2(v)$ and $f(u) = v$ because $\exists f. \forall u. set_1(u) \Rightarrow \exists v. set_2(v)$ and $f(u) = v$. We choose $\lambda z. x'$ for $f$. $\exists u. set_1(u) \Rightarrow \exists v. set_2(v)$ and $x' = v$ since $\exists u. set_1(u) \Rightarrow \exists v. set_2(v)$ and $\lambda z. x' = v$.

Let $set_1(u) \neq \exists v. set_2(v)$ and $x' = v$.

It isn’t the case that we have $set_2(x')$ and $x' = x'$. $x' = x'$. We have a contradiction since we have $set_2(x')$ and $x' = x'$. $\exists x. set_2(x)$ since we have a contradiction.

Although comparable textbook proofs might be much shorter we point out that considering the size of the actual ground-level proof it is already a rather concise and abstract presentation. Moreover the verbalization models the overall proof idea to derive a contradiction by instantiating $f$ depending on $x'$ as an element of $set_2$.

4 Using $\Omega$MEGA’s Knowledge Base

One of the features allowing TPS to automatically prove many theorems in higher order logic is the ability of selectively instantiating definitions. Indeed some theorems containing definitions can be proved with TPS using the dual instantiation strategy, whereas they cannot be automatically proven when all definitions are fully expanded [4].

In $\Omega$MEGA mathematical knowledge is structured into a hierarchy of theories where a theory can inherit from one or several parent theories. Each theory contains declarations of signature, axioms and definitions, where the latter can be viewed as abbreviations of more complex concepts. Moreover tactics, planning methods, linguistic knowledge and control strategies guiding the planner can be associated with each theory. Theorems are always declared within the context of a theory and can be presented concisely when using given definitions.
Fig. 7. The grounded proof tree

In order to enable TPS to prove certain subgoals containing concepts unknown to TPS it would be necessary to expand these definitions completely before sending the problem to TPS. Yet this would not only mean that the respective formulas might grow to a size intractable by TPS but we would also prevent TPS from using its mechanism for selectively instantiating definitions. Thus it is necessary to transfer definitions of used concepts from $\Omega$MEGA to TPS. This can be achieved by using TPS’ built-in library mechanism.

While in $\Omega$MEGA all objects, i.e. axioms, definitions, theorems, tactics, etc., a priori are associated with an existing theory, in TPS theories are created in order to group objects — thereby also creating hierarchies by specifying one theory as object of another — but a single object does not necessarily depend on a theory. Therefore we can map $\Omega$MEGA definitions onto corresponding TPS abbreviations by either

(A) transferring single concepts together with all related definitions and axioms into the TPS library, or by
(B) constructing a TPS library that mirrors both hierarchical structure and objects of the $\Omega$MEGA knowledge base.

So far we have implemented an ad hoc version of approach (A). The underlying algorithm is stated in Fig. 8. However, we believe that with an extension of this algorithm goal (B), i.e. the transfer of the whole $\Omega$MEGA knowledge base into
a TPS library, can also be achieved. With the algorithm new definitions are stepwise expanded and inserted, together with their underlying concepts, into the library.

1. get all concepts used in the ΩMEGA formula
2. for each concept C do:
   - if concept C already exists in the TPS library proceed with next concept
   - if it does not yet exist, then:
     • get definition of C from the ΩMEGA knowledge base
     • extract all concepts C₁, . . . , Cₙ occurring in definition of C
     • retrieve all axioms A₁, . . . , Aₘ referring to C from the ΩMEGA knowledge base and insert them as formulas into the TPS library
     • insert TPS abbreviation of C into the TPS library specifying the dependency with respect to C₁, . . . , Cₙ and A₁, . . . , Aₘ
     • apply step 2 to {C₁, . . . , Cₙ}
   • proceed with next concept

**Fig. 8.** Algorithm for transferring ΩMEGA definitions into TPS libraries

In order to demonstrate the working scheme of the algorithm and exemplify the translation of concepts, we consider the example theorem

\[ \forall X, \emptyset \in \mathcal{P}(X). \tag{1} \]

Conjecture (1) contains three definitions from ΩMEGA's naive-set theory. Explicitly these are \emptyset, \epsilon, and \mathcal{P}, where \emptyset and \epsilon are defined as in Sec. 2 and \mathcal{P}, specifying the powerset of a set X, can be expressed as the polymorphic \lambda-term:

\[ \mathcal{P} = \lambda X. \emptyset \cup \lambda Y. Y \subseteq X \]

As both \emptyset and \epsilon do not depend on any further definitions or axiomatization they can directly be translated into the TPS library. The definition of \mathcal{P} however depends on \subseteq which expresses the concept that Y is a subset of X. Therefore the abbreviation for powerset in the TPS library is defined with respect to the

\[ ^5 \text{ A translation of the whole ΩMEGA knowledge base into a TPS library should certainly be done explicit once only, instead of automatically whenever a new theorem is transferred.} \]
The concept of subset, which in turn fetched from ΩMEGA’s knowledge base. There Ξ is defined by
\[ Ξ = \lambda X_0 \lambda Y_0 \lambda Y \lambda Z_0 X \implies Y(Z) \]
and, as it neither contains any other abbreviations nor there exist axioms referring to it, can be translated into the TPS library.

5 Conclusion and Future Work

We reported about the integration of the higher order theorem prover TPS into the mathematical assistant system ΩMEGA. As a result TPS can now be used to either automatically or interactively construct proofs, which then get translated and integrated into ΩMEGA’s proof data structure. The tactic based proof transformation approach makes use of a special ΩMEGA theory TPS providing ΩMEGA-tactics for each possible TPS-justification. These tactics can subsequently be expanded onto ΩMEGA’s calculus level, thereby enabling the graphical representation and the verbalization of TPS proofs on different levels of abstraction. In order to use TPS’ reasoning power most efficiently we presented a way of translating and importing facts from ΩMEGA’s knowledge base into TPS libraries. All these features have been presented in this paper with small examples.

We admit that the translation of RuleP by mapping it to a call of the first order theorem prover OTTER is somewhat an overkill. Even if it costs only a little additional time and is done fully automatically, we should at least intercept very trivial cases by mapping those immediately to appropriate tactics or rules. Furthermore we should replace OTTER by a pure propositional logic prover such as SATO [21] as soon as it is integrated in ΩMEGA. Nevertheless we want to point out that this translation strategy demonstrates an interesting feature of ΩMEGA in general: All external reasoners already integrated in ΩMEGA can be used to support the integration of new systems, e.g. to close gaps when translating protocols of the new systems. Consequently as more specialized systems are being integrated with ΩMEGA the less detailed protocols may be required from the new systems.

The integration work we have done so far is essentially restricted to modifications of the ΩMEGA system and it seems to be promising to modify TPS as well, probably enabling a bidirectional communication between the two systems. As TPS also provides a tactic mechanism it seems to be plausible that an analogous translation from one of ΩMEGA’s abstract proof levels into a TPS proof can be developed. Thereby external reasoners integrated to ΩMEGA as well as its internal facilities, would automatically become available to TPS. Surely TPS does not provide a three-dimensional proof data structure and hence ΩMEGA proofs cannot be mirrored in their original form within TPS, but this should not impose a serious problem for the cooperation of both systems.

\[ * \] Both powerset and subset are already built-in abbreviations of TPS. Yet we used them here for the sake of simplifying the examples.
Although the question about an intelligent automatic cooperation between both systems is open a combined system will at least gain additional deductive power when used in an interactive mode where the user controls the overall strategy. The authors currently work on an integration of higher order theorem provers into OMEGA's proof planning framework. Examples, such as proving the equivalence of two different mathematical definitions (e.g. group definitions), demonstrate that both systems have complementary strengths: Especially the decomposition of such proof problems into subtasks can easily be handled by OMEGA’s proof planning framework, whereas most of the particular subtasks might be solvable automatically by TPS, when using its mechanism for selectively instantiating the definitions. Currently neither OMEGA nor TPS is able to prove such theorems fully automatically, but their combination most likely is. Certainly the development of an advanced common theory and library mechanism will be an important prerequisite.

Two other interesting questions are: First, if TPS’ automatic search process and OMEGA’s built-in logical engine LEO [7], which is a resolution based higher order theorem prover specialized in an appropriate treatment of the extensionality principles, could cooperate successfully. And second, whether the combination of both higher order theorem provers could be a even more promising support for OMEGA’s proof planner.

Closely related work has been done independently from the work presented here in combining the interactive theorem prover HOL with the CLAM proof planner [18]. Similar to our approach they translate proof plans generated by CLAM first into an abstract tactic level and then into the concrete HOL tactics. However, since the logics of both systems are quite different the work concentrates on the complicated syntax mapping between the systems. In contrast to this, syntax transformation do not play such a big role in our work as the logics of OMEGA and TPS are very much alike. Therefore different concepts in the two systems need not to be mapped onto each other in a static way but instead can be easily translated and used by the other system.

6 Acknowledgements

We thank Matthew Bishop and Peter Andrews for supporting our integration work and for providing us with important information about TPS. Furthermore we thank Holger Gebhard for his aid with the implementation and Serge Autuxier for stimulating remarks.

References


Some Theorem Proving Aids

Paul E. Black\(^1\)\(^*\) and Phillip J. Windley\(^2\)

\(^1\) National Institute of Standards and Technology  
Gaithersburg, Maryland 20899, USA  
paul.black@nist.gov

\(^2\) Computer Science Department, Brigham Young University  
Provo, Utah 84602-6576, USA  
windlecy@cs.byu.edu

Abstract. Theorem proving can be a very useful formal method. However, it currently takes a lot of time and study to learn how to use a theorem prover, and proving even apparently simple theorems can be tedious. Theorem proving, and its benefits in software and hardware development, should be accepted more readily and widely if new users can do larger proofs of more complete models earlier in their training and with less work. We present some generally applicable tools which we found helpful in formally verifying a secure web server. The first is a program to check goals for common mistakes arising indirectly from type inference. We also give tactics, or proof advancing routines, to simplify goals and handle assumptions. Finally we give tactics which prove goals by selecting assumptions to establish the goal or find a contradiction. These are another step to making theorem proving easier, increasing productive, and reducing unnecessary complication.

1 Introduction

Although proving theorems with mechanized support can be useful in many industrial developments, using a theorem prover can take a great deal of expertise. Many people consider theorem proving to be unrealistic because of the time to learn how to do proofs and the tedium of proofs. However, more powerful tactics would allow users to “work at a high enough level to make the proof process practical” \[^3\].

Unfortunately powerful tactics tend to be slow. In fact, Gödel’s theorem assures us there is a limit to the power of automatic tactics \[^5,6\]. However as computers have gotten faster, it makes sense to automate more and more theorem proving, even using heuristics which may be time-consuming or not guaranteed to work. \texttt{MESON_TAC} \[^4\] is an example. HOL traditionally has concentrated on efficient “building block” tactics rather than strong tactics as, say, PVS \[^7\] or Isabelle \[^8\].

\(^*\) This work was sponsored in part by the National Science Foundation under NSF grant MIP-9412581
We describe general tactics and a program we developed while verifying a secure web server [1]. Section 2 presents a small program to help analyze why a rewrite may fail on a particular goal. Section 3 gives several tactics to simplify goals. The last set of tactics, in Sect. 4, attempt to prove goals by grouping assumptions for other conversions. (Early versions of these sections are in [1].) Finally we report our use of these tactics and our conclusions in Sect. 6.

2 A Program to Analyze Rewrite Failures

Our first aid to theorem proving is a small program to analyze rewrite failures. One may make mistakes in assigning types or may misenter names, especially when developing specifications or formalizations. These errors may be especially frustrating when a rewrite fails for no apparent reason. We wrote a routine in SML to help analyze these situations.

WHY_NOT() examines the current, or top, goal and reports possible subtleties which may be preventing it from being proved. It searches for and reports the following situations.

1. Variables or constants with the same name, but different types.
2. Variables or constants with the same type and similar names. That is, possible typographical errors.
3. A variable and a constant with the same name and type.
4. Variables whose names are valid constants for that common types.

Such subtle differences are hard to spot and have wasted a lot of users’ time finding them. Since types in HOL are often inferred, identifiers with the same name, but different types, are rare. But because HOL typically does not print types, the user may have a hard time finding the problem when it does occur. Also with the type inference, it is not unheard of to mistype a variable or constant and not notice the problem for some time. For example, one researcher typed a goal similar to the following.

\[(\text{empty}_q \Rightarrow \text{done}) \land (\neg \text{done} \land \text{started} \Rightarrow \neg \text{empty}_q)\]

Boolean types were inferred for all variables. It took several frustrating hours trying to prove the goal before realizing that \text{empty}_q had been mistyped as \text{empty}_q.

Even more difficult to find is if a constant and a variable have the same name and type. There is printed indication that something is a constant instead of a variable. Finally it is possible to create a numeric variable named 42 or a string variable named "k". Again there is no way to distinguish these from constants except using the predicates is const or is var.
The implementation is straightforward. First all atoms (variables or constants) are extracted from the top goal and assumption list. Numeric constants, short string constants, and constants whose names are common operators, such as $\lor$ or $\&$, are ignored. The “type” check reports atoms with the same name, but different types. The “spelling” check reports atoms which identical types and similar names. Names are similar if the initial character is the same and the rest differs by a single character replacement, deletion, or insertion or a single transposition. We require the initial characters to match to avoid reporting pairs like $x\text{Size}$ and $y\text{Size}$. Names shorter than three characters are ignored, too, to reduce false reports. The “kind” check reports atoms with the same name and type where one is a constant and the other is a variable. Finally list of atoms is checked for variables which are numerals, strings, or common operators.

The following is a contrived, small example. The intent is to set a goal to prove $x + 2 + abc = abc + 1 + y + 1$ given $x = y \land abc > 5$. However abc is mistyped as abe. More seriously since $=$ has lower precedence than $\land$, the hypothesis is associated as $x = (y \land (abc > 5))$, so $x$ and $y$ are type boolean instead of natural numbers. The errors are pretty obvious here, but these kinds of errors are much harder to catch when the predicates are big or there are lots of assumptions. The goal looks like it could be proved by rewriting and arithmetic analysis ($\text{ASM\_REWRITE\_TAC} \;$ and $\text{CONV\_TAC ARITH\_CONV}$), but the “errors” prevent it.

\begin{verbatim}
- set_goal([x = y \land abc > 5],
  x + 2 + abc = abe + 1 + y + 1);

val it =
  Initial goal:
  x + 2 + abc = abe + 1 + y + 1

  x = y \land abc > 5

- WHY\_NOT();
\end{verbatim}

The name $y$ appears as both $:\text{num}$ and $:\text{bool}$
The name $x$ appears as both $:\text{num}$ and $:\text{bool}$
Possible typo: abe and abc have the same type and similar names

Types in HOL are prefaced with a colon ($:\text{num}$ and $:\text{bool}$). The user prompt is a dash and space ($-\;$). We give more details about HOL in App. A.

3 Tactics to Simplify Goals

Often the theorem to be proved is so complex it is hard for the novice to know where to start. These complex goals may arise in discharging obligations of theorems, so it may not even be immediately clear what the theorem means. This section gives tactics we have found helpful to simplify goals and take small, but significant, steps toward proving them. The tactics may even prove the goal.
3.1 General Simplification

Proofs in axiomatic semantics tend to carry conjunctions of many conditions. Inference rules often involve one extended condition implying another where most of the conditions can be trivially satisfied. So there are often goals similar to $a \land b \land c \land D \Rightarrow a \land b \land c \land E$.

Sometimes a goal can be proved just by stripping quantifiers and implications, then rewriting with assumptions. We combine these steps into one tactic which we call STRIP_THEN_REWRITE_TAC. It is simply

```ml
val STRIP_THEN_REWRITE_TAC =
  REPEAT STRIP_TAC THEN ASM_REWRITE_TAC [];
```

This may prove the goal or leave any number of subgoals. It never fails, but since it does general rewriting, it may not terminate. It has always terminated in practice. We give an example in Sect. 3.4.

Even when this doesn’t prove the goal, it usually clarifies the goal by reducing it to a number of simpler subgoals. When there is a problem with the proof, for instance, missing an assumption, this disentangles what needs to be proved. If some tactic would be helpful, say expanding a definition, this can be a diagnostic aid by showing what needs to be proved and the conditions or assumptions. One can “back up” or undo the invocation, apply the necessary tactic, and proceed.

3.2 Move Quantifiers Outward

Complex inference rules may leave deeply buried quantifiers in the goal. It can be hard to even determine the scope of quantification. LIFT_QUANT_TAC combines all available conversions to move universal and existential quantifiers as far outward as possible. There universal quantifiers can be stripped and existential quantifiers can have witnesses provided in one step, rather than encountering them at different, odd times in the proof. Here is the definition.

```ml
val LIFT_QUANT_TAC =
  CONV_TAC (REDEPTH_CONV (AND_EXISTS_CONV ORELSEC AND_FORALL_CONV ORELSEC
  OR_EXISTS_CONV ORELSEC OR_FORALL_CONV ORELSEC
  LEFT_AND_EXISTS_CONV ORELSEC LEFT_AND_FORALL_CONV ORELSEC
  LEFT_IMP_EXISTS_CONV ORELSEC LEFT_IMP_FORALL_CONV ORELSEC
  LEFT_OR_EXISTS_CONV ORELSEC LEFT_OR_FORALL_CONV ORELSEC
  RIGHT_AND_EXISTS_CONV ORELSEC RIGHT_AND_FORALL_CONV ORELSEC
  RIGHT_IMP_EXISTS_CONV ORELSEC RIGHT_IMP_FORALL_CONV ORELSEC
  RIGHT_OR_EXISTS_CONV ORELSEC RIGHT_OR_FORALL_CONV));
```
This tactic never fails. We have used it after undischarging all assumptions to simplify all of them at once. See Sects. 3.3 for an example.

### 3.3 Undischarge a Selected Assumption

Handling assumptions can be difficult, especially if one is trying to write a reusable proof. Proofs are less sensitive to change if assumptions are selected by a predicate or filter rather than by exact match or position. (A program to generate filters from an assumption list is given in [2].) `FILTER_UNDISCH_TAC` undischarges an assumption which matches an arbitrary predicate.

```ml
val FILTER_UNDISCH_TAC fp =
  let fun hfp t = fp (concl t) handle _ => false
  in
    ASSUM_LIST (fn th1 =>
      UNDISCH_TAC ((concl o hd) (filter hfp th1)))
  end;
```

This raises an exception if no assumption matches. The user supplies a term predicate to the tactic. For instance, the following undischarges the first assumption where the right hand side (#rhs) of an equality (dest_eq) is 0, that is, \(\ldots = 0\). (Term delimiters are `--' and `--' in HOL.)

```ml
e (FILTER_UNDISCH_TAC (fn t => (#rhs o dest_eq) t = (--'0'--')));
```

The following example comes from a proof of information integrity. The filter function looks for an assumption which is universally quantified.

```ml
prefSS inode (getFile SYS_fileSystem' inode)

\forall inode.
    (inode = inode0f (deref "FP)) \lor
    prefSS inode (getFile SYS_FileSystem' inode)
\neg (inode = inode0f (deref "FP))

- e (FILTER_UNDISCH_TAC (fn t => is forall t));
```
1 subgoal:
(∀ inode.
  (inode = inode0f (deref 'FP))\n  preFSS inode (getFile SYS_FileSystem' inode)) \implies
preFSS inode (getFile SYS_FileSystem' inode)

¬(inode = inode0f (deref 'FP))

The following tactic proves the goal.

e (LIFT_QUANT_TAC THEN EXISTS_TAC (--‘inode:num’--) THEN
   ASM_REWRITE_TAC []);

3.4 Undischarge All Assumptions

The tactic UNDISCH_ALL_TAC undischarges all assumptions. It is helpful when
one needs to manipulate all the assumptions at once. Here is the definition.

val UNDISCH_ALL_TAC =
  REPEAT (FIRST_ASSUM (fn thm => UNDISCH_TAC (concl thm)));

This tactic leaves the original goal, but with all assumptions undischarged.
The following extended example comes from the proof of confidentiality of a
call to fprintf(). We are proving that the precondition (from the previous
statement’s postcondition) implies the required precondition for fprintf(). This
example also shows the use of LIFT_QUANT_TAC and STRIP_THEN_REWRITE_TAC.

nonConfidential (getFile SYS_FileSystem' SYS_stdout)

FP = 'FP
¬(inode0f (deref FP) = SYS_stdout)
∀ inode, (inode = SYS_stdout) \implies
   nonConfidential (getFile SYS_FileSystem inode)
C_Result16 > 0
∀ inode, (¬(inode = inode0f (deref FP)))\n  \lor
  (¬prev, (inode = SYS_stdout) \implies
   nonConfidential prev)\n  \lor
  (appendFile (printfSpec "%s %s %s %d" vargs)
   prev = getFile SYS_FileSystem' inode))\n  \lor
  ((inode = inode0f (deref FP))\n   \lor
   (inode = SYS_stdout) \implies
   nonConfidential (getFile SYS_FileSystem' inode)))
_inode = SYS_stdout

¬ e (UNDISCH_ALL_TAC);
1 subgoal:
(\texttt{FP} = \texttt{"FP"}) \Rightarrow \neg (\texttt{inodeOf (dereF FP) = SYS\_stdout}) \Rightarrow
(\forall \texttt{inode} = \texttt{SYS\_stdout}) \Rightarrow
nonConfidential (getFile SYS\_FileSystem \texttt{inode})) \Rightarrow
C\_Result9 > 0 \Rightarrow
(\forall \texttt{inode} \neg (\texttt{inode} = \texttt{inode0f (dereF FP)}) \lor
(\exists \texttt{prev}, (\texttt{inode} = \texttt{SYS\_stdout}) \Rightarrow nonConfidential \texttt{prev}) \land
(appendFile (printfSpec "\%s \%s \%s \%s \%d" \texttt{vargs})
prev = getFile SYS\_FileSystem' \texttt{inode}))) \land
((\texttt{inode} = \texttt{inode0f (dereF FP)}) \lor
((\texttt{inode} = \texttt{SYS\_stdout}) \Rightarrow
nonConfidential (getFile SYS\_FileSystem' \texttt{inode}))) \Rightarrow
(inode = \texttt{SYS\_stdout}) \Rightarrow
nonConfidential (getFile SYS\_FileSystem' SYS\_stdout)

- \texttt{e \ (LIFT\_QUANT\_TAC)};

1 subgoal:
\exists \texttt{inode'} \texttt{inode''},
\forall \texttt{prev}, (\texttt{FP} = \texttt{"FP"}) \Rightarrow \neg (\texttt{inodeOf (dereF FP) = SYS\_stdout}) \Rightarrow
((\texttt{inode'} = \texttt{SYS\_stdout}) \Rightarrow
nonConfidential (getFile SYS\_FileSystem \texttt{inode'}))) \Rightarrow
C\_Result9 > 0 \Rightarrow
(\neg (\texttt{inode''} = \texttt{inode0f (dereF FP)}) \lor
((\texttt{inode''} = \texttt{SYS\_stdout}) \Rightarrow nonConfidential \texttt{prev}) \land
(appendFile (printfSpec "\%s \%s \%s \%s \%d" \texttt{vargs})
prev = getFile SYS\_FileSystem' \texttt{inode''})) \land
((\texttt{inode''} = \texttt{inode0f (dereF FP)}) \lor
((\texttt{inode''} = \texttt{SYS\_stdout}) \Rightarrow
nonConfidential (getFile SYS\_FileSystem' \texttt{inode''}))) \Rightarrow
(inode = \texttt{SYS\_stdout}) \Rightarrow
nonConfidential (getFile SYS\_FileSystem' SYS\_stdout)

3.5 An Arithmetic Tactic

While trying to prove theorems about array accesses, we came across goals which appeared easy, but took quite a bit of work. HOL has a number of good, basic tactics for arithmetic, but we could not find any general tactics.

DEPTH\_ARITH\_TAC simplifies as many arithmetic expressions as possible. This may prove the goal, but even if it doesn’t, it eliminates some subexpressions so the user can concentrate on the parts which are not proved automatically. The implementation is as follows.

val DEPTH\_ARITH\_TAC = REDUCE\_TAC THEN
CONV\_TAC (ONCE\_DEPTH\_CONV
    (ARITH\_CONV ORELSEC NEGATE\_CONV ARITH\_CONV)) THEN
ONCE\_REWRITE\_TAC [];}
The following example is ripped from a proof that a piece of code finds the maximum in an array. (Some conjuncts were removed to make the example more readable.) Since \( n \) is modeled as a natural number, \texttt{DEPTH\_ARITH\_TAC} eliminates the \( 0 \leq n \) clauses.

\[
(j \leq \text{arSz}\wedge \\
(\forall n. 0 \leq n \wedge n < j \Rightarrow \\
\quad \text{max} \geq \text{CA\_IDX}(\text{CA\_FN ar}\text{arSz}n))\wedge \\
\quad j \geq \text{arSz} \Rightarrow \\
(\forall n. 0 \leq n \wedge n < \text{CA\_SZ ar} \Rightarrow \text{max} \geq \text{CA\_IDX ar} n)
\]

\texttt{DEPTH\_ARITH\_TAC} is somewhat inefficient since for every expression, it tries to prove that the expression is true with \texttt{ARITH\_CONV}, then if that fails, that it is false with \texttt{NEGATE\_CONV ARITH\_CONV}.

4 Tactics to Prove Goals Using Assumptions

Many times goals can be proved by instantiating assumptions or finding contradictions among assumptions. Since it may be difficult to work with assumptions in HOL [2], we find it helpful for a program to try combinations of assumptions. The three tactics in this section provide a general way to automatically manipulate assumptions to prove a goal. The first two tactics, \texttt{ESTAB\_TAC} and \texttt{INCONSIST\_TAC} are somewhat specific but share the same mechanism. The last one, \texttt{SOLVE\_TAC}, builds on the first two for a more general tactic.

4.1 Establish a Term From the Assumptions

\texttt{ESTAB\_TAC} adds the term operand as an assumption if it is provable from the assumptions. The core is a utility, \texttt{establish}, which returns a theorem of the form \( a_1 \wedge \ldots \wedge a_n \Rightarrow tm \). If \texttt{establish} can prove an appropriate theorem, \texttt{ESTAB\_TAC} uses it to add the term. This tactic fails if the term cannot be added.

Given a term and list of assumptions, \texttt{establish} tries different combinations of assumptions to prove the term. It tries \texttt{ARITH\_CONV} and \texttt{TAUT\_CONV} to prove the resulting theorem. For efficiency, \texttt{establish} tries to prove the term from each assumption, then pairs of assumptions, then triples. It only tries pairs and triples of assumptions if they share free variables.
Suppose you have the following goal.

\[ j > 0 \]

\[ j \geq \text{ar} \text{Sz} \]
\[ \forall n \, n < j \Rightarrow \text{max} \geq f \, n \]
\[ \text{max} = 0 \]
\[ \text{ar} \text{Sz} > 0 \]
\[ P \, \text{ar} = \text{ar} \text{Sz} \]

Some inspection shows that we could establish \( j > 0 \) from \( j \geq \text{ar} \text{Sz} \) and \( \text{ar} \text{Sz} > 0 \). The following tactic proves the above goal.

\[ \text{e} (\text{ESTAB_TAC} \ j > 0); \]

Without \text{ESTAB_TAC} the proving tactic is considerably longer, more sensitive to changes, and less clear.

\[ \text{e} (\text{IMP_RES_TAC} (\text{prove} (j \geq \text{ar} \text{Sz} \land \text{ar} \text{Sz} > 0 \land j \leq \text{ar} \text{Sz} \Rightarrow j > 0, \text{ARITH_TAC}))); \]

4.2 Find an Inconsistency in the Assumptions

\text{INCONSIST_TAC} tries to prove a goal by finding an inconsistency in the assumptions. It fails if it cannot prove the goal.

The implementation is to try to establish \( F \) (false) (see \text{ESTAB_TAC} for details), then prove the goal since false implies anything (using \text{CONTR_TAC}). If it cannot establish \( F \), it adds assumptions which follow from equalities (e.g., \( a = b \)) and other assumptions until it finds a matching inequality (e.g., \( \sim (a = b) \)), which is an inconsistency. The idea and implementation of this second approach is due to Robert Beers (beers@lal.cs.byu.edu).

Consider the following goal. Inspection suggests a proof by contradiction using the assumptions \( n < j \) and \( j = 0 \) since \( n \) and \( j \) are natural numbers. \text{INCONSIST_TAC} proves this goal.

\[ \text{someFunction} \ n = 0 \]

\[ n < j \]
\[ 0 \leq n \]
\[ j = 0 \]

4.3 Prove Several General Ways

The tactic \text{SOLVE_TAC} heavily uses \text{ESTAB_TAC} and \text{INCONSIST_TAC} to solve a goal. It tries a series of approaches and specialized tactics to prove the goal. We chose the approaches from situations which arose in proving software properties. The different ways are:

1. Establish the goal from the assumptions.
2. Prove an inconsistency in the assumptions.
3. If the goal is $a = b$, establish equalities to unify $a$ and $b$.
4. If an assumption is $a \Rightarrow b$, establish $a$ and the unifiers for $b$ and the goal.

It fails if it cannot prove the goal.

SOLVE_TAC proves the following goals automatically.

\[
\begin{align*}
\text{P d} \\
\text{a} \\
\text{a }\Rightarrow\text{ P c} \\
\text{c }=\text{ d}
\end{align*}
\]

\text{nonConfidential (getFile SYS.FileSystem SYS.stdout)}

\[
\begin{align*}
\text{inode }=\text{ SYS.stdout} \\
\forall \text{inode} (\text{inode }=\text{ SYS.stdout}) \Rightarrow \\
\text{nonConfidential (getFile SYS.FileSystem inode)}
\end{align*}
\]

Because of their modular construction, these tactics can easily be improved. For instance, a version of SOLVE_TAC could take user's conversion, in the spirit of variations on REWRITE_TAC. Also SOLVE_TAC can be extended to handle a broader class of goals and assumptions and could memoize its attempts to unify. The establish routine could be more selective about which groups of assumptions to try and also do more preprocessing.

5 Experience

Our verification of a secure web server [1], consists of a total of about 720 tactic invocations in about 2,600 lines of tactics and comments. (tac1 THEN tac2 counts as two invocations.) About 17% of the invocations are STRIP_THEN_REWRITE_TAC, and 6% are our other new tactics; details are in Table 1.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Tactic & Number of uses \\
\hline
STRIP_THEN_REWRITE_TAC & 120 \\
SOLVE_TAC & 20 \\
FILTER_UNDISCH_TAC & 10 \\
UNDISCH_ALL_TAC & 5 \\
LIFT_QUANT_TAC & 2 \\
\hline
\end{tabular}
\caption{Uses of tactics}
\end{table}

Although efficiency was not the goal, these tools are quick. On an HP 9000 the largest goals in our verification, which are over two hundred printed lines,
took under two seconds for \texttt{WHY\_NOT} to analyze. \texttt{DEPT
HARITH\_TAC} took less than eight seconds on the largest goals. The example in Sect. 3.4 with invocations of \texttt{SOLVE\_TAC} takes about 2.5 seconds.

Sources are available at the following URL. \texttt{ESTAB\_TAC} and \texttt{INCONSIST\_TAC} are in establish.sml, and \texttt{SOLVE\_TAC} is in solveTac.sml. The file \texttt{whynot.sml} contains \texttt{WHY\_NOT}. The rest of the tactics are in utilities.sml.

http://hissa.ncsl.nist.gov/~black/Source/

6 Conclusions

We have presented several generally applicable theorem proving tools. The tools include a program to analyze goals for problems which may not be caught because of HOL's type inference, tactics to simplify goals, and tactics to automatically pick out assumptions to advance a proof.

Arguably none of these tools is a breakthrough, but the group of them automates many tedious parts of proofs. They also provide a framework for incremental work to make incremental improvements in more automated proofs. These tools help to make theorem proving a little easier for new or casual users, reduce the amount of learning needed to get results, and handle details so the user can do proofs at a slightly higher level.

Acknowledgement

We thank Robert Beers for his inspiration.

References


A  HOL Syntax and Conventions

The following shows how a goal is displayed.

\[
\begin{align*}
&\exists d P(d) \\
&\text{a \& klear'} \lor \neg qq \\
&\forall \text{inode} (\text{inode} = \text{SYS.stdout}) \Rightarrow \\
&\quad \text{nonConfidential} (\text{getFile SYS.FileSystem inode})
\end{align*}
\]

The goal is printed above the line, and all hypotheses are printed below it. Types are by default not displayed. However the user can turn on type printing.

Unbound variables are assumed to be universally quantified. Names may include underscores (_ ) and primes ( ‘ ). Antiquotation or program (SML) variable interpolation is introduced by a caret ( ^ ).

Function application is implied (no parentheses are needed), and functions may be curried. For instance, getFile SYS.FileSystem inode means the function getFile applied to arguments SYS.FileSystem and inode.
Verification of the MDG Components Library in HOL

Paul Curzon\textsuperscript{1}, Sofiène Tahar\textsuperscript{2}, and Otmane Aït Mohamed\textsuperscript{3}

\textsuperscript{1} School of Computing Science, Middlesex University, London, UK
\texttt{p.curzon@mdx.ac.uk}
\textsuperscript{2} ECE Department, Concordia University, Montreal, Canada
\texttt{tahar@ece.concordia.ca}
\textsuperscript{3} IRO Department, University of Montreal, Canada
\texttt{ait@iro.umontreal.ca}

Abstract. The MDG system is a decision diagram based verification tool, primarily designed for hardware verification. It is based on Multiway decision diagrams—an extension of the traditional ROBDD approach. In this paper we describe the formal verification of the component library of the MDG system, using HOL. The hardware component library, whilst relatively simple, has been a source of errors in an earlier developmental version of the MDG system. Thus verifying these aspects is of real utility towards the verification of a decision diagram based verification system. This work demonstrates how machine assisted proof can be of practical utility when applied to a small focused problem.

1 Introduction

Verification systems can themselves contain errors. In the worst case this could result in a faulty application being certified correct. Ideally verification systems should themselves be formally verified: preferably using a verification system with a different architecture. In general, this is not practical, as verification systems are very large pieces of software. However, it can still be useful to verify aspects of the system, even if a full verification is not completed. In this paper we investigate the verification of the component library of a decision diagram based verification system using the HOL theorem prover [11]. The verification system under investigation is the MDG system [5]. This is a real hardware verification system that has been used in the verification of significant hardware examples [2]. It consists of a simple wide-spectrum hardware description language (MDG-HDL) in which both structural and behavioral hardware descriptions can be written. These descriptions are converted to an internal decision diagram representation, upon which the verification is performed. A fundamental primitive of the hardware description language is the table. In its simplest form this is just a truth table representation of a relation between the values on variables. Used with don’t-care and default values, next state variables and variable entries it becomes a powerful specification construct that can be used to give behavioral specifications of hardware as abstract state machines (ASM) [5].
Tables are also used internally in the MDG implementation. They provide a simple and uniform means of implementing other primitive components. The current implementation of the MDG system provides a library of basic components in addition to the table with which hardware can be described. Examples include flip-flops and logic gates. Many of these primitives are implemented internally as tables. We have verified this library of components, proving that the table versions implemented in the MDG system are equivalent to the desired semantics of the components as specified in higher-order logic.

The library is only a small part of the MDG system. However, it is critically important that the components are correctly implemented. The MDG system provides a range of verification tools, including property checking, equivalence checking and reachability analysis. Each of these make use of the library primitives. For example, properties which abstractly can be thought of as temporal logic formulae are written in (or translated to) the HDL and thus make use of the library components.

One of the motivations for our work was an error in a table representation of one component, the JK flip-flop with enable. This error was discovered in a developmental version of the system, found during the actual verification of a hardware design [12]. The system was erroneously indicating there was an error in the design being verified. The erroneous component had only recently been added to the system specifically because it was needed for the verification of the hardware design [12] (only a JK flip-flop variant without enable was available within the library). This error was corrected in the system prior to our work. We have demonstrated that the new version is correct and that the other components implemented as tables are also correct. Furthermore, we have provided precise formal specifications of each library component. Finally we have provided simple parameterized HOL tactics which can be used to automatically verify future additions to the library.

2 Related Work

There has been a variety of techniques used to ensure the correctness of verification systems. In the LCF approach [9], also used in the HOL system, an abstract data type of theorem is used to ensure that only a core of functions corresponding to the primitive inference rules and axioms of the logic can compromise the system. All derived rules call these primitives to create theorems. Thus the validity of proved theorems is guaranteed by the type system of the implementation language, provided the primitives are correct.

A second approach experimented with in the HOL system by von Wright [13] and Wong [14], was that of independent proof checking. In this approach, the main verification system produces a log of the primitive inference steps used in a proof. This log can then be checked by an independent proof checker. Such a checker has to include only implementations of the primitive rules. It can thus be much simpler than a full theorem prover and is thus less likely to contain errors. Such a proof checker has been implemented for the HOL system by Wong [14].
Due to its simplicity, verifying such a proof checker is also more tractable. Von Wright demonstrated this by verifying the specification of a proof checker for the HOL system against a formal semantics of the HOL logic [13]. This specification was also used by Wong as the specification for his implementation, thus increasing the confidence in its correctness. A problem with this approach, however, is that the proof scripts generated are very large and the time taken to check a real proof may be intractable.

Other work on the verification of verification systems includes that of Homeier and Martin [8] who used the HOL system to verify a verification condition generator for a simple programming language. Chou and Peled [4] similarly used HOL to verify a partial-order reduction technique used to reduce the state-space exploration performed by model checkers. The technique examined is used in the SPIN system. This was a significant proof effort, resulting in almost 7500 lines of proof script and taking 10 weeks to complete.

3 MDG System

3.1 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) have been proposed recently [5] as a solution to the data width problem of ROBDD based verification tools. The MDG tool combines the advantages of representing a circuit at higher abstract levels as is possible in a theorem prover, and of the automation offered by ROBDD based tools. MDGs, a new class of decision graphs, comprises, but is much broader than, the class of ROBDDs [1]. It is based on a subset of many-sorted first-order logic, augmented with a distinction between abstract and concrete sorts. Concrete sorts have enumerations which are sets of individual constants, while abstract sorts do not. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. Data operations are represented by uninterpreted function symbols.

An MDG is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of a concrete sort with its edge labels the individual constants in the enumeration of the sort. It can also be a variable of abstract sort with its edges labeled by abstract terms of the same sort. Finally, it can be a cross-term (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted as T, which means all paths in the MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs incorporate variables of abstract type to denote data signals and uninterpreted function symbols to denote data operations. MDGs can also represent sets of states. They are thus much more compact than ROBDDs for designs containing a datapath. Furthermore, sequential circuits can be verified independently of the width of the datapath.
MDGs are used as the underlying representation for a set of hardware verification tools, providing both validity checking and verification based on state-space exploration. The MDG tools package the basic MDG operators and verification procedures [16]. The operators are disjunction, relational product (conjunction followed by existential quantification) and pruning-by-subsumption. The verification procedures are combinatorial and sequential verification. The combinatorial verification provides the equivalence checking of two combinational circuits. The sequential verification provides invariant checking and equivalence checking of two state machines. The MDG operators and verification procedures are implemented in Quintus Prolog [16].

3.2 MDG-HDL

The MDG tools accept as hardware description a Prolog-style HDL, MDG-HDL [16], which allows the use of abstract variables and uninterpreted function symbols. The MDG-HDL description is then compiled into the internal MDG data structures. MDG-HDL supports structural descriptions, behavioral descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG-HDL) connected by signals. A behavioral description is given by a tabular representation of the transition/output relation. The tabular constructor is similar to a truth table but allows first-order terms in rows. It allows the description of high-level constructs as ITE (If-Then-Else) formulas and CASE formulas.

A circuit description includes the definition of signals, components and the circuit outputs. Signals are declared along with their sorts, e.g. signal(x, wordn), where x is a signal of an abstract sort wordn. Components are declared by the instantiation of the input/output ports of a predefined component module. For example, a multiplexer with a control signal select of concrete sort having [0, 1, 2, 3] as an enumeration, inputs: x0, x1, x2, x3 of an abstract sort α, and output: y of the same abstract sort α is defined as:

```prolog
component(mux1, mux(select(select),
       inputs([[0,0),(1,x1),(2,x2),(3,x3)]),
       output(y))
```

Besides circuit descriptions, a variety of information, such as sort and function type definitions, symbol ordering and invariant specification, etc., have to be provided in order to use the applications outlined above.

As part of the MDG software package, the user is provided with a large set of predefined modules such as logic gates, multiplexers, registers, bus drivers, etc. Besides the logic gates which use Boolean signals, all other components allow signals with concrete as well as abstract types. Among predefined modules we have a special module called a table. Tables can be used to describe a functional block in the implementation, as well as in the specification. A table is similar to the truth table, but it allows first-order terms in the rows. A table is essentially a series of lists, together with a single final default value. The first list contains
variables and cross-terms. The last element of the list must be a variable (either concrete or abstract). The other variables in the list must be concrete variables. The remaining lists consist of the sets of values that the corresponding variables or cross-terms can take. The last element in the list of values could be a first-order term. This represents an assignment to the output variable. The other values must be either “don’t cares” (represented by ‘?’) or individual constants in the enumeration of their corresponding variable sort. The last element in a table is the default value. It is a term giving the value of the output variable when a set of values arises that is not explicitly given in the table. Fig. 1 illustrates different representation of an and gate with two inputs $x_1, x_2$ and one output $y$. Fig. 1(b) shows the MDG-HDL declaration of this gate using the primitive component and. The behavior of this gate can be described as a table (Fig. 1(c)) which can be written in MDG-HDL as follows:

$$\text{table}([[x_1, x_2, y], [0, *, 0], [1, 0, 0], [1, 1, 1]])$$

This table description is further internally translated into an MDG (decision diagram) with the variable ordering $x_1 < x_2 < y$ (Fig. 1(d)).

![Gate and MDG-HDL](image)

**Fig. 1.** Different representations of an and gate

A further example of the use of tables with abstract variables and functions is given by the following example:

$$\text{table}([[c, \text{leq}(x, y), n_y], [1, 1, x] | y])$$

which defines the function

$$\text{if } (c = 1) \text{ and } (\text{leq}(x, y) = 1) \text{ then } n_y = x \text{ else } n_y = y.$$  

where $c$ is a concrete boolean variable, $x$ is an abstract input variable, $y$ is an abstract state variable, and $n_y$ represents its next state. $\text{leq}$ represents a function symbol that means “less-or-equal”. The term $y$ after symbol ‘|’ in the table description is used as the default value.
4 Formalizing the MDG Library in HOL

The first step in the verification is to give formal specifications of the library components to be verified. This is a relatively simple task, since the components are mainly logic gates and flip-flops. Traditional relational hardware semantics in the style of Gordon [10] can be given. Signals are represented as functions from time (a natural number) to the value at that time. The semantics of a component is then a relation between the input signals and the output signals. For example, the \texttt{and} gate would be specified as:

\[
\text{AND } x_1 \times x_2 \ x = \forall(t:\text{num}). \ y \ t = (x_1 \ t) \land (x_2 \ t)
\]

Here \( y \) is the output signal and \( x_1 \) and \( x_2 \) are the input signals. Similar specifications are given for each component in the library to be verified, as well as for tables. The definition for tables is more complex, requiring recursive definitions.

4.1 MDG-Tables

A table can be thought of as taking 5 arguments. The first argument is a list of the inputs, the second is the single output, the third is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The entries in the list can be either actual values or a special don't-care marker. The latter matches any value the input could hold. The fourth argument is a list of output values. Each is the value on the output when the inputs have the values in the corresponding row. The final argument is the default value, taken by the output if the input values do not match any row.

Thus for example the \texttt{and} gate, specified above could be represented by the arguments:

\[
([x_1, x_2], y, [[0,0],[0,1],[1,0],[1,1]], [0, 0, 0, 1], -)
\]

The inputs are \( x_1 \) and \( x_2 \), the output is \( y \), the possible values for the inputs are \( (0,0), (0,1), (1,0) \) and \( (1,1) \). The corresponding values on the output are \( 0, 0, 0 \) and \( 1 \), respectively. Here no default value is needed as all cases are covered. An alternative version, making use of the don't-care value (given by \( * \)), is

\[
([x_1, x_2], y, [[0,*],[1,0],[1,1]], [0, 0, 1], -)
\]

A more compact version still using the default value would be:

\[
([x_1, x_2], y, [[1,1]], [1], 0)
\]

If both inputs are \( 1 \) then so is the output, otherwise the output is \( 0 \). Similarly, the MDG system's implementation of a \texttt{JK} flip flop with enable is the table:
([e, j, k, q1, nq, 
[[0,0,0,0], 
[0,0,1,1], 
[1,1,0,0], 
[1,1,0,1], 
[1,0,1,0], 
[1,0,1,1], 
[1,0,0,0], 
[1,0,0,1], 
[1,1,0,0], 
[1,1,0,1], 
[1,1,1,1]], 
[0,1,1,0,0,1,1,0], -)

Here, e is the enable signal, q represents the last output and nq the next output.

Our HOL specifications are based on the above representation. In fact the implementation which is in Prolog, uses a slightly different representation, taking a single list of list argument and a further default value. The first version of the and gate above actually appears in the implementation as the following (with no default specified).

[[x1, x2, y], [0,0,0],[0,1,0],[1,0,0],[1,1,1]]

Here, the inputs and output appear in a single list, with the latter distinguished by its position. In our description above and our HOL treatment we have separated out the components for clarity of definition. It should be noted that the above representation could not be used in our HOL treatment given below, as the lists have different types: values as opposed to traces of values (including don’t-care) over time. It is thus possible that we could have made transcription mistakes from one form to the other. However, it would be relatively simple to modify our table definition to use two arguments: a variable list and a row list in the same order as in the MDG implementation. This would merely involve adding a wrapper function to the TABLE definition, which extracted the appropriate arguments.

4.2 Table Formalization in HOL

The first step in formalizing this definition is to define a type for table values. These can be either a normal value of arbitrary type or a don’t-care value. This is defined as a new HOL type, with associated destructor function to access the value.

Table_Val = TABLE_VAL of 'a | DONT_CARE

TableVal_to_Val (TABLE_VAL (v:'a)) = v

We next define the matching of input values to table values. A match occurs if either the table value is don’t-care, or the value on the input is identical to the table value. This property must hold for each table entry. It is defined recursively by a function table_match.
\text{(Table\_match inputs } \Box(t:\text{num}) = T) \wedge \
\text{(Table\_match inputs } (\text{CONS } v \text{ vs}) t = 
\begin{array}{l}
((\text{HD}(\text{inputs}) \ t) = \text{TableVal\_to\_Val (v:'a Table\_Val) } \lor \\
(v = \text{DONT\_CARE}) \wedge \\
(\text{Table\_match (TL inputs) vs t}))
\end{array}
\]

If there is a match on a given row, the output has the corresponding value. Otherwise, we must check the next row. If there is no match, the output equals the default value. This is defined recursively on the input list as the relation table:

\text{(table inps (out: num } \rightarrow \ 'b) (\Box:('a Table\_Val list) \ V\_out default t = 
\begin{array}{l}
(\text{out } t = \text{default } t) \wedge 
\end{array}
\]

\text{(table inps out (CONS } v \text{ vs) } V\_out default t = 
\begin{array}{l}
((\text{Table\_match inps } v \ t) \Rightarrow \\
(\text{out } t = (\text{HD } V\_out) t) \mid \\
(\text{table inps out vs (TL } V\_out) \text{ default } t)))
\end{array}
\]

The above definitions refer to the time of interest, \( t \). A given table will relate a given input to a given output, if the table relation is true at all times:

\text{TABLE inps (out: num } \rightarrow \ 'b) (V\_outs:('a Table\_Val list) \ V\_out default = 
\begin{array}{l}
\forall t. \text{table inps out } V\_outs V\_out \text{ default } t
\end{array}
\]

The above relation \text{TABLE}, thus defines the semantics of an MDG table. Using the HOL notation the Table for the AND component would be specified as:

\text{AND\_TABLE } x1 x2 y = 
\begin{array}{l}
\text{TABLE } [(x1:\text{num } \rightarrow \text{bool}); x2(y: \text{num } \rightarrow \text{bool})] \\
[[\text{TABLE\_VAL } F; \text{TABLE\_VAL } F]; \\
[\text{TABLE\_VAL } F; \text{TABLE\_VAL } T]; \\
[\text{TABLE\_VAL } T; \text{TABLE\_VAL } F]; \\
[\text{TABLE\_VAL } T; \text{TABLE\_VAL } T]]
\end{array}

[FSIG;FSIG;FSIG;TSIG] TSIG

We use the HOL booleans \text{F} and \text{T} for 0 and 1, respectively. Note that the values given in the input rows and default value are not values but signals; that is, functions from time to a value. The constant signals for 0 and 1 are thus represented by \text{TSIG} and \text{FSIG} which are just lifted versions of the constants.

\text{FSIG} = \lambda(t:\text{num}). \ F \\
\text{TSIG} = \lambda(t:\text{num}). \ T

The definition that we give is less flexible than the MDG system’s tables since all the input values are restricted to be of the same type, whereas in the MDG system they can be of a variety of sorts. In the next subsection we present a way to deal with this problem.
4.3 Application of the Table Definition to Multisorts Inputs

In the above formalization of the MDG tables, it is assumed that the inputs of the table are of the same type. This is true for most components (gates) of the MDG-HDL library. In order to represent the MDG table of a more general component with inputs of different types in HOL, we need to extend our formalization to accommodate a list of inputs (the first argument of the table definition) with different types. As an example we present the formalization of the state transition diagram of the timing block of the Fairisle ATM switch fabric [6] in terms of an MDG table in HOL. Fig. 2 shows the finite state machine of the behavior of this timing block, which consists of three symbolic states (RUN, WAIT, ROUTE), and has two inputs (frameStart and anyActive) and one output routeEnable.

![State transitions of the Fairisle switch fabric timing block](image)

The MDG table of the next state function of this state machine is:

```
[[anyActive, frameStart, timing_state, n_timing_state],
[*1, run, wait],
[*0, run, run],
[1, 0, wait, route],
[*0, route, run],
[*1, route, wait]] | wait
```

While the inputs and the output are of boolean sort, timing_state and n_timing_state are of a concrete sort with the enumeration: RUN, WAIT, ROUTE. We hence need to create a common type for all the input variables as well as the state variable timing_state in order to use our definition of tables in HOL.

Let TIMING_TYPE_VAL be the states of our machine:

```
TIMING_TYPE_VAL = RUN | WAIT | ROUTE
```

The common type for all the input variables, TIMING_SPEC_TYPE, is defined as:
TIMING_SPEC_TYPE = TRANS of ‘a | STATE of TIMING_TYPE_VAL

Having these ingredients, we derive the HOL definition of the above table as:

TABLE [anyActive;frameStart;timing_state](timing_state o NEXT)
[DONTCARE;TABLE_VAL(TRANS T);TABLE_VAL(STATE RUN)];
[DONTCARE;TABLE_VAL(TRANS F);TABLE_VAL(STATE RUN)];
[TABLE_VAL(TRANS T);TABLE_VAL(TRANS F);TABLE_VAL(STATE WAIT)];
[DONTCARE;TABLE_VAL(TRANS F);TABLE_VAL(STATE ROUTE)];
[DONTCARE;TABLE_VAL(TRANS T);TABLE_VAL(STATE ROUTE)]
[WAITSIG;RUNSIG;ROUTESIG;RUNSIG;WAITSIG] WAITSIG

where RUNSIG, WAITSIG, ROUTESIG, are lifted versions of the constants RUN, WAIT and ROUTE.

RUNSIG = λ(t:num).(STATE:TIMING_TYPE_VAL -> bool TIMING_SPEC) RUN
WAITSIG = λ(t:num).(STATE:TIMING_TYPE_VAL -> bool TIMING_SPEC) WAIT
ROUTESIG = λ(t:num).(STATE:TIMING_TYPE_VAL -> bool TIMING_SPEC) ROUTE

4.4 Formal Verification of the Library Components

To verify a library component, we must prove that the semantics of the table used in the MDG implementation is equivalent to the semantics of the component. For example, for the and component we prove the theorem:

∀x1 x2 y. AND x1 x2 y = AND_TABLE x1 x2 y

This can be proved easily in HOL by first rewriting with the definitions and then applying the recently added, efficient tactic MESON_TAC. This was packaged into a simple tactic, COMB_MDG_TAC, that was then used to prove all combinational components in the library. For sequential components such as the RS flip-flop and JK flip-flops with and without enable, we use a different tactic, SEQ_MDG_TAC, based on rewriting and cases analysis, that we parameterize with respect to the input variables.

5 Use of Results

5.1 MDG Components Library

The main result of this work is that we have verified all components of the MDG component library except a few that are not implemented in terms of tables. This gives increased confidence in the MDG system. The work was originally motivated by an error found in a table in an early version of the system. This error was introduced because a new component (a JK flip-flop with enable) was added to the system on the fly. It is likely that new components will be added in the future. Tables provide a flexible and convenient way for this to be done. However, as they consist of tables of 1’s and 0’s it is easy to make
mistakes. Our HOL theory and automatic proof tool, provide a simple, fast and convenient method for such future additions to be formally verified. As for the library components, this consists of giving the formal specification of the component in HOL, writing the Table definition in HOL, setting the goal and applying the tactic. The proof will of course only be automatic for simple components of the level of complexity found in the existing library. Users of the MDG system are liable to want to define their own similar primitive components. They can use the theory and proof tool in the same way. We have thus provided a toolkit (albeit limited) for both users and developers of the MDG system.

5.2 HOL Tables Theory

A definition and associated theory of tables is a useful addition to HOL in its own right, as tables provide a flexible means of giving definitions of logic functions. The definition we used in our proofs is not suitable directly as a general definition, as it has an explicit notion of time $t$ built in: this was most convenient for our application as we did wish to include a time component in our definitions. A more general definition of a table would have thus unnecessarily complicated the final definitions and proofs.

A more suitable definition for general use would be:

$(\text{Tab\_match} \ \text{inputs} \ [\cdot] = T) \land$

$(\text{Tab\_match} \ \text{inputs} \ (\text{CONS} \ v \ \text{vs})) =$

$((\text{HD} \ \text{inputs} = \text{TableVal} \ \text{to} \ \text{Val} \ (v: \ \text{a TableVal})) \lor$

$(v = \text{DON'T\_CARE}) \land (\text{Tab\_match} \ (\text{TL} \ \text{inputs}) \ \text{vs}))$

$(\text{TAB} \ \text{insps} \ (\text{out}: \text{'b}) \ ([\cdot]: \text{('a TableVal \ \text{list}) \ \text{list}) \ V_{\text{out}} \ \text{default} =$

$(\text{out} = \text{default}) \land$

$(\text{TAB} \ \text{insps} \ \text{out} \ (\text{CONS} \ v \ \text{vs}) \ V_{\text{out}} \ \text{default} =$

$((\text{Tab\_match} \ \text{insps} \ v)) \Rightarrow$

$(\text{out} = (\text{HD} \ V_{\text{out}})) \lor$

$(\text{TAB} \ \text{insps} \ \text{out} \ \text{vs} \ (\text{TL} \ V_{\text{out}} \ \text{default}))$)

5.3 Formal Verification of the MDG System

Some of the library components such as the multiplexer are implemented directly in terms of MDGs, rather than as a table that is then implemented as an MDG. The other such components are the register (with and without a control input), fork (equality of signals), transform (for uninterpreted functional blocks), and drivers (essentially a guarded command). However, in theory all the components could be implemented as tables. If this were done, these components could be verified in the same way as the ones we considered here. Then, the correctness of the library would depend only on the correctness of the translation of the tables into MDGs, rather than on the way a series of components were implemented as MDGs. It is also worth noting that tables have a fairly simple translation into a basic MDG.
Proposed Source Languages

VHDL / FSM

MODULAR

MDG-HDL

ACTL

FLATTENED

MDG-HDL

TABLES

MDG

Fig. 3. Format translations within the MDG system

The above proofs correspond to the first step in a larger project to verify a formal specification of the MDG system [15]. The system can be considered as a series of translators, translating between different intermediate languages, as shown in Fig. 3. One step in that process is the translation from the MDG-HDL language to a subset of the language with only tables as components. This table subset is then translated into MDGs in a series of further translation steps. Currently structural, behavioral and property specifications are all given in this low level language. However, translators from specialist higher level languages are under development as shown in Fig. 3.

The correctness of a translator between two languages can be stated in terms of the semantics of the languages, as shown in Fig. 4. Essentially this states that the translation should preserve the semantics of the source language. This is the traditional form of compiler specification correctness used in the verification of compilers [3]. The same approach can be used to specify and verify a hardware verification system such as MDG. For the translation to tables the correctness theorem would have the form

\[ \forall h. \ S_h(h) = S_t(T(h)) \]

where \( h \) is a hardware description, \( S_h \) is the semantics of the source language, \( S_t \) is the semantics of the table subset and \( T \) is a functional specification of the translation between the two. The proof of this theorem proceeds by structural induction on the source language. It requires lemmas stating that the translation of each kind of hardware component is correct. These lemmas are in fact the theorems that we have proved above.

5.4 MDG-HOL Hybrid System

The current work is also of relevance to a further project: namely that of combining the MDG and HOL systems, to give a hybrid hardware verification tool. The verification of a 16 by 16 switch fabric, already verified in the pure HOL system [7] is being used as a case study in this project. If results from the MDG system are to be imported into HOL, then the structural and behavioral specifications used must have HOL equivalents. The hardware semantics used in the work described here provides such a basis. Using the same semantics in the
6 Summary and Conclusions

We have formally specified the semantics of the MDG component library using HOL. This includes a formalization of the Table construct that forms the heart of the MDG wide-spectrum hardware description language. We then formally described the table implementations of each of the hardware components that are implemented in terms of tables in the MDG system. We verified the correctness of each table implementation against the formal specification of the component. This was done using two simple automated tactics in HOL, one for combinational library components and one for sequential ones. These tactics were sufficiently flexible and powerful to verify all table-based components of the MDG library.

We have thus proved the correctness of one small but crucial part of the MDG system, thus increasing the confidence of users of MDG have of the system. Whilst the table implementations we verified are a relatively simple part of the system, errors have previously been uncovered in such table definitions. Our verification is thus of practical utility.
We have demonstrated how a theorem prover can be of utility on real and
highly complex software, if a small and well-defined problem is tackled. Fur-
thermore, by verifying a decision diagram system using a verification system
implemented on a different paradigm, we have reduced the possibility that the
verification is flawed due to an error in the verification system used.

We have also given formal specifications of the library components, which
will help ensure users have an accurate understanding of those components and
so use them correctly. The automated HOL proof tool can be used by system de-
signers to ensure that new components added to the MDG system are correctly
implemented. Similarly users of MDG who need to define their own basic com-
ponents in terms of tables can use the HOL proof tool to ensure the correctness
of those tables.

Finally, the work done forms the first step of a larger project to verify a formal
specification of the MDG system. The theorems proved are the main lemmas that
would be needed in verifying one stage of such a formal specification. Similarly,
the hardware semantics given for the components are an essential step in the
ongoing project to combine the HOL and MDG systems. By using the same
semantics for both these projects we open the way for linking the correctness
proof of MDG with hybrid proofs of hardware using the combined system. We
have also paved the way towards providing a toolkit that can be used by both
the users of the combined MDG system and developers of that system.

References

1. R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE

2. E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. Automated
Verification with Abstract State Machines Using Multiway Decision Graphs. In T.
Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*,

3. L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness
Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–
214, 1986.

4. C.-T. Chou and D. Peled. Formal Verification of a Partial-Order Reduction Tech-
nique for Model Checking. In T. Margaria and B. Steffen, editors, *Tools and Algo-
rithms for the Construction and Analysis of Systems*, Lecture Notes in Computer

5. F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision
Graphs for Automated Hardware Verification. *Formal Methods in System Design*,

6. P. Curzon. The Formal Verification of the Fairisle ATM Switching Element: an
Overview. Technical Report no. 328, University of Cambridge, Computer Labora-

of the International Conference on Engineering of Complex Computer Systems*,
Simulating Term-Rewriting in LPF and in Display Logic

Jeremy E. Dawson

Department of Computer Science and Automated Reasoning Project
Australian National University, Canberra ACT 0200, Australia
jeremy@arp.anu.edu.au

Abstract. We show how the convenience and power of term-rewriting can sometimes be obtained in logical systems which do not explicitly have this capability. We consider the Logic of Partial Functions, and show how an undefined term can often be rewritten to a defined term. Although LPF and Display Logic are unrelated, we also show how Display Logic effectively allows rewrite-style simplifications, although the logic has no axiom or rule permitting this (or indeed any notion of equality). We then describe how these “rewrite” procedures are implemented in Isabelle, using HOL-style conversionals.

Keywords: term rewriting, logic of partial functions, undefined terms, display logic

1 Introduction

The convenience of proof by term-rewriting is demonstrated by the theorem provers which rely wholly or primarily upon it (eg Larch [13]), and by the prominent place that rewriting tactics have in provers such as Isabelle [18] and HOL [9].

The Logic of Partial Functions (LPF) handles undefined terms, and is the logic underlying VDM (see [4]). It has been mechanized in Isabelle by Agerholm & Frost [1]. Proofs in LPF could use standard rewriting of equal terms, but this would be complicated by the need to prove that the term to be rewritten is defined, since an undefined term is not equal to anything. In Sect. 2 we demonstrate that this complication can be avoided in LPF by using not equality but the partial order ⊆ of domain theory, or “replaceability”, as a condition for rewriting one term by another.

Display Logic [3] is a generalization of the Gentzen sequent calculus. Several logics have been formalized in it. Like the sequent calculus, Display Logic does not have an explicit notion of equality of subterms. Therefore, one might imagine that term rewriting would not be available in Display Logic. The equivalence relation ≡ which arises from the standard Lindenbaum algebra (as in [12, Sect. 4]) is usually a congruence, as is normally required for rewriting. However this relation is not available in Display Logic itself, since it is a meta-level concept. In Sect. 3 we discuss tactics which, effectively, enable rewriting in Display Logic.

In Sect. 4 we consider the features of these logics which make these tactics possible. We looked for a common thread between the two methods used (for
LPF and for Display Logic), without success. Rather, Sect. 4 shows a contrast between the two methods.

The source files containing the ML functions described are available at

http://arp.anu.edu.au:80/~jeremy/TPHOLs98

2 Logic of Partial Functions

2.1 Introduction

Partial functions – which can yield “undefined” results – are common in computing, and proving results about programs requires a method of handling undefined terms. In [3] and [16] various methods of either avoiding or coping with them are discussed. The Logic of Partial Functions, described in [2], is one of these methods. It is a three-valued logic, where propositional terms can be true, false or undefined. The title of [16] expresses the conclusion that every method has disadvantages; in relation to LPF, in [2, page 264] the authors state “Even given the basic set of axioms, it was not immediately apparent how to avoid clouding proofs with case distinctions concerning undefined”. In this section we show how this complication may often be avoided.

LPF is described in [2], where its semantics are given, together with a sound and complete Hilbert-style axiom system. Briefly,

- the semantics are based on three truth-values, true (t), false (f) and undefined (⊥).
- \( p \lor q \) is true if either \( p \) or \( q \) is true, false if both \( p \) and \( q \) are false, and undefined otherwise.
- \( \sim p \) is defined precisely when \( p \) is; \( \sim t \) is \( f \) and \( \sim f \) is \( t \).
- the other logical connectives can be defined by the usual identities, ie, \( p \land q \) is \( \sim (\sim p \lor \sim q) \) and \( p \rightarrow q \) is \( \sim p \lor q \).
- the quantifiers \( \forall x \) and \( \exists x \) mean “for all defined \( x \)” and “there exists a defined \( x \).”
- \( \Delta x \) means “\( x \) is defined”; \( \Delta x \) can only be true or false.
- a related operator is \( \delta \), where \( \delta x \) is \( x \lor \sim x \); thus \( \delta x \) is undefined exactly when \( \Delta x \) is false.
- a valid derivation gives a conclusion which is true when the premises are all true; it may give any conclusion when a premise is undefined or false (in particular, an undefined premise may give a false conclusion).

Our \( \rightarrow \) is written \( \Rightarrow \) in [2]. We use \( \rightarrow \) to avoid confusion with the Isabelle convention, which we use, where \( P \Rightarrow Q \) means “\( P \) can be derived from \( Q \)” (for which [2] uses \( \vdash \)). We also use \( P \rightarrow Q \) to mean the rule “rewrite \( P \) to \( Q \).”

Thus all the logical operators behave as in classical logic when their operands are defined. Furthermore, many usual identities of classical logic, such as the associativity and commutativity of \( \lor \), and that \( \sim \sim p \) is identical to \( p \), hold. Many others, involving \( \land \) and \( \rightarrow \), hold since these operators can be defined in terms of \( \lor \) and \( \sim \).
We now introduce the partial order \( \sqsubseteq \) in which \( \perp \sqsubseteq f \) and \( \perp \sqsubseteq t \), \( t \) and \( f \) being incomparable. This is the domain-theory partial order (see [17], Ch. 5, in which \( \omega \) is used for \( \perp \)). All the operators above, except \( \Delta \), are “monotone” relative to this partial order. Other types, such as numbers \( \mathbb{N} \), are augmented with an undefined element \( \perp_{\mathbb{N}} \), and \( \sqsubseteq \) can be defined on each such type.

If we further define \( p \leftrightarrow q \) to be \( (p \rightarrow q) \land (q \rightarrow p) \), as usual, we get that \( p \leftrightarrow q \) is undefined whenever either \( p \) or \( q \) (or both) is undefined. Thus \( p \leftrightarrow p \) is undefined whenever \( p \) is. In LPF, ‘\( = \)’ denotes “weak equality” (whose operands may be non-logical terms, such as numbers), which behaves similarly to \( \leftrightarrow \); that is, \( x = x \) is true only if \( x \) is defined, and is undefined otherwise. With “strong equality”, denoted by ‘\( == \)’, \( x == y \) is also true if both \( x \) and \( y \) are undefined.

A definition (“let \( x \) equal \( y/z \)”)) is necessarily interpreted as strong equality.

If the undefined value is taken, intuitively, to mean “unknown” (may be true, may be false), then the truth tables for the monotone operators are consistent with this intuition. This is the basis of the notion of “replaceability” described below. However, because the values of expressions are determined by calculation using the truth tables, the expressions \( p \leftrightarrow p \) and \( \sim p \lor p \) (equivalently, \( p \rightarrow p \)) may be undefined (which is not consistent with this intuition, since under it both these examples are “known” to be “true”). In fact, taking an undefined term as referring to a real but unknown quantity is one of the other approaches described in [5] and [16].

2.2 Rewriting subterms

The fact that a valid derivation may give a false conclusion from undefined premises is the key to the results of this section. (In [2] this is called the weak interpretation of the semantic turnstile). It means that if we take a premise, and apply to it transformations which

- transform true propositions to true propositions
- transform undefined or false propositions to anything

then we have an inference which is valid in LPF. In fact we will use a sub-class of such transformations, namely those which

(a) “behave nicely” on defined propositions, i.e., transform them to equivalent propositions
(b) may “misbehave” on undefined propositions, i.e., transform them to anything

The value of this approach is that we can use a transformation (or rewrite rule) which does “misbehave” on undefined propositions, and yet avoid the need to check whether a proposition is defined.

We give an example based on division by zero, \( x/0 \) being undefined. At this point we treat \( x * 0 \) as undefined if \( x \) is undefined. Then the derivation 1/0 * 0 = 1/0 * 0 \( \Rightarrow \) 1 = 0 is valid in LPF, because the premise is undefined. (This may be thought a disadvantage of LPF, and is commented upon later). We obtain this derivation using the methods of this section as shown in Fig. 1: we
apply the rewrite rule $x/y * y \rightarrow x$ to the left-hand side, and apply the rewrite rule $z * 0 \rightarrow 0$ to the right-hand side. These rewrite rules are applied without having to check which terms are defined.

\[
\frac{1/0 * 0 = 1/0 * 0}{1 = 1/0 * 0} \quad \frac{(x/y * y \rightarrow x)}{(z * 0 \rightarrow 0)} 
\]

**Fig. 1.** A valid derivation in LPF

In the usual theory of numbers, the first rule above, if expressed as an equality, $x/y * y = x$, would not be valid; the second rule $z * 0 \rightarrow 0$ looks all right, but we apply it in a case where $z$ is undefined. We now look at the justification for proceeding this way.

We say a term $x$ is “replaceable” by another term $y$ either if $x$ is undefined or if $x$ is (defined and) equal to $y$. That is (in terms of the domain ordering) $x \sqsubseteq y$. Note that, if $x$ and $y$ are propositional term, this means that a transformation satisfying the description in (a) and (b) above may take $x$ to $y$.

Now, if $f$ is an operator monotone in its first argument, then, by definition, $f(x, \ldots) \sqsubseteq f(y, \ldots)$. We extend this to the following result.

**Lemma 1.** Let $x \sqsubseteq y$, and let $g(x, \ldots)$ be a propositional expression made up from $x$ (and other operand terms) using only monotone operators. Then the inference $g(x, \ldots) \implies g(y, \ldots)$ is valid.

**Proof.** Omitting reference to operands other than $x$, we can define $g(x) = f_n(f_{n-1}(\ldots(f_1(x))\ldots))$, where each operator $f_i$ is monotone. As $x \sqsubseteq y$ and $f_1$ is monotone, $f_1(x) \sqsubseteq f_1(y)$; generally, if $f_i(f_{i-1}(\ldots(f_1(x))\ldots)) \sqsubseteq f_i(f_{i-1}(\ldots(f_1(y))\ldots)$ and $f_{i+1}$ is monotone, then $f_{i+1}(f_i(f_{i-1}(\ldots(f_1(x))\ldots)) \sqsubseteq f_{i+1}(f_i(f_{i-1}(\ldots(f_1(y))\ldots))$. Therefore, by induction, $f_n(f_{n-1}(\ldots(f_1(x))\ldots) \sqsubseteq f_n(f_{n-1}(\ldots(f_1(y))\ldots))$: that is, $g(x, \ldots) \sqsubseteq g(y, \ldots)$. Finally, as $g(x, \ldots)$ is a proposition, then $g(x, \ldots) \implies g(y, \ldots)$ means that either they both have the same truth-value, or $g(x, \ldots)$ is undefined. A fortiori, if $g(x, \ldots)$ is true then so is $g(y, \ldots)$. Thus the inference $g(x, \ldots) \implies g(y, \ldots)$ is valid. \hfill \Box

Note that every operator which is “strict” or “naturally extended” (ie, an undefined operand produces an undefined result) is monotone ([17], p. 361).

Thus, when $x \sqsubseteq y$, this result allows rewriting $x$ by $y$ without checking whether $x = y$, ie without checking whether $x$ is defined. However the example shown in Fig. 1 illustrates points at which one must beware of the issue of undefinedness.

Firstly, it is important to remember that some axioms (such as $x = x$) and inference rules (such as implication-introduction) do require that some term be
defined. (Otherwise one could use the valid rule 1/0 * 0 = 1/0 * 0 \implies 1 = 0 to deduce 1 = 0). Secondly, care is needed in formulating the definitions of operators and the replacement rules. For example, if we chose to define \( x = 0 \) to be 0 whether or not \( x \) is defined (which would be quite sensible, since \( '=' \) would still be monotone), then the premise of the example in Fig. 1 would be true. However, with this definition the rewrite rule \( x/y + y \rightarrow x \) would not be valid, since \( x/y + y \) and \( x \) could be both defined but not equal, in which case \( x/y + y \not\equiv x \).

2.3 Implementation in Isabelle

Although Isabelle has a built-in rewriting capability, it is limited to the case where terms are equal (actually, related by Isabelle’s meta-equality operator). Since the rewriting described here ultimately uses instead inference steps of the form of (2) below, it is not possible to use Isabelle’s built-in rewriting.

The implementation of this technique was motivated by the manner in which the HOL theorem prover permits the user to specify a strategy for navigating through a term and rewriting rewriteable subterms where they are found. In HOL, this is done by specifying conversions and conversionals. A full description of these is in [9, Chap. 13]. A conversion is a function of type \texttt{term -> thm}, which takes a term \( t \) to a theorem \( t = t' \) (which can be used to rewrite \( t \) to \( t' \)).

A conversional is a function which acts on or modifies conversions. For example, if \( \text{conv} \ t \) is \( t = t' \) (to rewrite \( t \) to \( t' \)), and \( \text{conv} \ t' \) is \( t' = t'' \) (to rewrite \( t' \) to \( t'' \)) then \( \text{conv THENC} \ t = t'' \) (to rewrite \( t \) to \( t'' \)). Another example is the \texttt{REPEATC} conversional, which repeatedly applies a conversion until there is no further change to the term. These are described in [9, Sect. 13.1]. The conversional \texttt{SUB_CONV} applies a conversion to the immediate subterms of a term. These conversionals may be used to program various more complex strategies for rewriting (where possible) subterms of a term, as described in [9, Sect. 13.2].

A conversion \( \text{conv} \) that “fails”, in the sense that \( \text{conv} \ t \) finds no \( t' \) such that \( t = t' \), can be programmed either to return \( t = t \) or to raise an exception, and there are functions to change one sort of conversion to the other.

Any conversion can be converted to either a forward proof rule or a tactic for backwards proof by the functions \texttt{CONV_RULE} and \texttt{CONV_TAC}, see [9, Sect. 13.1]. The conversions form the basis of \texttt{REWRITE_RULE} and \texttt{REWRITE_TAC}, used in forward proof and backward proof respectively.

Implementation of the simulated rewriting technique for LPF relies heavily on the concepts of the HOL implementation, though the details differ.

In Isabelle, we implemented the axioms and rules of [2] as a theory. In addition, we defined a function \( \text{rep}(t,t') \) to mean that \( t \) is replaceable by \( t' \), (ie, \( t \sqsubseteq t' \)) and we proved, from the LPF axioms, results of the form

\[
\text{rep}(P,P') \quad \text{rep}(Q,Q') \\
\frac{\text{rep}(P \ op \ Q, P' \ op \ Q')}{\text{rep}(P, P')} 
\]

where \( op \) stands for each of the binary propositional operators, and corresponding results for the unary propositional operators. We also proved the reflexivity
and transitivity of \texttt{rep}, and the following result

\[
\frac{\texttt{rep}(P, Q)}{P} \frac{P}{Q}
\]

(2)

We define a new datatype which provides a way (other than by using exceptions) to indicate whether or not a term is unchanged,

\[
\texttt{datatype} \; \texttt{crep} = \texttt{Unc} \mid \texttt{Rep} \; \texttt{of} \; \texttt{thm} ;
\]

A \texttt{rep-conversion}, of type \texttt{cterm} \rightarrow \texttt{crep}, is given a term \texttt{t} and produces a theorem. In the case of a rep-conversion used for forwards (resp. backwards) proof, the theorem is \texttt{rep}(t, t') (resp. \texttt{rep}(t', t)) for some t'. This is analogous to a HOL conversion, except that because \texttt{rep} is a preorder, not an equivalence, different rep-conversions are needed for forwards and for backwards proof.

A related function which helps in the implementation (in particular, it makes composition straightforward) is a \texttt{rep-transformation}, of type \texttt{thm} \rightarrow \texttt{crep}; a rep-transformation takes a theorem \texttt{rep}(s, t) and transforms it to the theorem \texttt{rep}(s, t') (for forwards proof) or \texttt{rep}(s', t) (for backwards proof). It does this by discovering the theorem \texttt{rep}(t, t') or \texttt{rep}(s', s) respectively, and using the transitivity of \texttt{rep}. In either case the resulting theorem is prefixed by \texttt{Rep}; if the function does not find \texttt{t'} (s') distinct from \texttt{t} (s) then \texttt{Unc} is returned (in which case we say the function \texttt{fails}). The transitivity and reflexivity of \texttt{rep} enables direct translation between rep-conversions and rep-transformations; the detailed implementation uses both.

We then define operators on these as follows:

\texttt{THENtt} : (\texttt{thm} \rightarrow \texttt{crep}) \ast (\texttt{thm} \rightarrow \texttt{crep}) \rightarrow \texttt{thm} \rightarrow \texttt{crep} (infix) applies two \texttt{rep-transformations} (of which the first need not succeed) in sequence.

\texttt{SUBcr} : (\texttt{cterm} \rightarrow \texttt{crep}) \rightarrow \texttt{cterm} \rightarrow \texttt{crep} applies a \texttt{rep-conversion} to the immediate subterms of a term. It uses previously proved rules of the form of (1) to "join" the results of applying the \texttt{rep-conversion} to the subterms.

\texttt{SUBtt} : (\texttt{thm} \rightarrow \texttt{crep}) \rightarrow \texttt{thm} \rightarrow \texttt{crep} corresponds to \texttt{SUBcr}, but is in terms of \texttt{rep-transformations}.

\texttt{REPtt} : (\texttt{thm} \rightarrow \texttt{crep}) \rightarrow \texttt{thm} \rightarrow \texttt{crep} repeats a \texttt{rep-transformation} one or more times until it \texttt{fails}.

These operators are for forward proof; of them, \texttt{SUBtt} needs to be programmed differently for backwards proof.

Various search strategies can now be defined easily, using the above operators. For example, \texttt{TDto tt}, given by

\[
\texttt{fun TDtt tt th = (REPtt tt THENtt (SUBtt (TDtt tt))) th ;}
\]

applies the \texttt{rep-transformation} \texttt{tt} repeatedly wherever possible, proceeding "top-down" through the term, first dealing with the whole term, then the immediate subterms, and so on down to the smallest subterms.

The following functions make \texttt{rep-transformations}:
rep tt : thm -> thm -> crep From a theorem rep(A, B), rep tt gives the
rep-transformation which transforms a theorem rep(X, A') to rep(X, B'),
where A' and B' are instantiations of A and B.

rep tt rev : thm -> thm -> crep is the corresponding function for backward
proof.

Finally, these rep-transformations can be applied in forwards or backwards
proof. There are functions to apply a rep-transformation to a theorem, in per-
forming forward proof, and to apply a rep-transformation to a subgoal as a tactic
in backwards proof.

3 Display Logic

3.1 Introduction

A number of different logical systems can be formulated using the method, or
style, of Display Logic [3]. These include several normal modal logics [21], and
intuitionistic logic [11]. Display Logic resembles the Gentzen sequent calculus
LK, but with significant differences. For example, the rule for introducing the
connective '∨' (on the left) in LK and its counterpart in Display Logic are

\[
\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \lor B \vdash \Delta} \quad (\text{LK-} \lor \vdash) \quad \frac{A \vdash Z \quad B \vdash Z}{A \lor B \vdash Z} \quad (\text{DL-} \lor \vdash) \quad (3)
\]

Whereas, in LK, \( \Gamma \) and \( \Delta \) denote comma-separated lists of formulae, in Display
Logic, \( Z \) denotes a Display Logic structure. A structure is defined as a formula, or
a combination of structures formed using the structural operators (one of which
is \( \cdot \)). Informally, formulae and formula (or logical) operators are the formulae
and operators of the logic being displayed, whereas the structural operators are
additional operators used in presenting that logic as a display logic. (See [12]
or [6] for a full explanation of structures and formulae). In Display Logic, unlike
in LK, the introduced formula (here, \( A \lor B \)) stands by itself on one side of the
turnstile; this is generally the case with Display Logic rules. However there are
also rules (the “display postulates”) which effectively allow moving substructures
from one side to the other. All display postulates are bi-directional (invertible)
rules.

In Display Logic, in terms of backwards proof, the “logical introduction rules”
eliminate the logical (formula) operators and replace them with correspond-
ing structural operators. For example, rules for introducing the \( \lor \) operator are
shown. Note that the \( (\lor \vdash) \) rule shown below is different from the one above; in
some Display Logics they both hold.

\[
\frac{A \vdash X \quad B \vdash Y}{A \lor B \vdash X, Y} \quad (\lor \vdash) \quad \frac{Z \vdash A, B}{Z \vdash A \lor B} \quad (\vdash \lor) \quad (4)
\]

A display logic will also have some “basic structural rules”, which are ex-
pressed in terms of structural connectives only, and which capture properties of
the logic and its logical operators. For example, the following rules express the associativity of $\land$ and $\lor$ (the double line means the rule is bi-directional).

$$
\frac{X, (Y, Z) \vdash W}{(X, Y), Z \vdash W} (A \vdash) \quad \frac{W \vdash X, (Y, Z)}{W \vdash (X, Y), Z} (\vdash A)
$$

As an alternative to writing a rule with one premise $\mathcal{P}$ and a conclusion $\mathcal{C}$ separated by a horizontal line, we often write $\mathcal{P} \Rightarrow \mathcal{C}$, and for a bi-directional rule we often write $\mathcal{P} \iff \mathcal{C}$. For a full explanation of Display Logic see, for example, [12].

By way of example, the Display Logic formulation of classical propositional logic has structural operators $\lnot$, $\ast$ $\dagger$ $\downarrow$ $\uparrow$. As in LK, $\lnot$ is used to stand for either $\lnot \land$ or $\lnot \lor$; which one it is depends not only on which side of the $\vdash$ the $\lnot$ appears, but also on the number of $\ast$ operators in whose scope the $\lnot$ lies. This reflects the duality between $\land$ and $\lor$, as expressed by DeMorgan’s laws. Each structural operator stands for one, or two (depending on the position), formula operators. Thus $\dagger$ stands for truth or falsity, $\ast$ for Boolean negation and $\uparrow$ for $\land$ or $\lor$. The logical identity $A \land B \Rightarrow C \iff A \Rightarrow C \lor \lnot B$ (which is an example of “residualization”, explained below) becomes an instance of the display postulate $X, Y \vdash Z \iff X \vdash Z, \ast Y$ and $\lnot A \Rightarrow B \iff \lnot B \Rightarrow A$ becomes an instance of $\ast X \vdash Y \iff \ast Y \vdash X$.

### 3.2 Residuation

As noted in Sect. 4, Display Logic does not have equalities or inverses; the fact that any chosen substructure can be displayed depends on the notion of residuation.

Consider a partially ordered set, with binary functions $s$, $f$ and $g$. To say that $f$ and $g$ are residuals of $s$ means that the following hold (for all $a$, $b$ and $c$)

$$
s(a, b) \leq c \iff a \leq f(c, b)
$$

$$
s(a, b) \leq c \iff b \leq g(a, c)
$$

(see [7], Sect. 6, or [10]). These equivalences have the effect of “displaying” $a$ and $b$ respectively. When all the connectives used in a proposition have residuals, any subterm can be displayed. If we then have a rule which partly instantiates to $a \leq X \Rightarrow a' \leq X$, we can form a proof as follows.

$$
s(a, b) \leq c \\
a \leq f(c, b) \\
a' \leq f(c, b) \\
s(a', b) \leq c
$$

We have in effect rewritten the subterm $a$ (located in an arbitrary position) to $a'$. This is the manner in which we rewrite arbitrarily chosen substructures in Display Logic.
Note that for the proof above, we required only \( a \leq X \implies a' \leq X \), not \( a \leq X \iff a' \leq X \). That is, \( a' \) may be a strictly stronger proposition than \( a \). Effectively, doing subterm rewrites this way also allows a subterm to be replaced by one that is logically either stronger or weaker, according to the subterm’s position.

### 3.3 Rewriting

It will be noted that among the examples above, both the basic structural rules \((A \vdash \top)\) and \((\vdash \neg A)\), and the logical introduction rule \((\vdash \lor)\) have on one side a single structural variable (which can stand for any structure, including any formula) which remains unchanged; the expression being “rewritten” appears on the other side. This is typical of many rules. In some other cases, rules of this form can be derived. For example, if the rule for \((\lor \vdash \top)\) given in the logic is that shown at (4), it may be possible to derive the version given at (3).

The following example shows this rule being used. The first (bottom) step is to display the disjunction \( A \lor C \) (“dp” denotes “display postulate(s)"); then the \((\lor \vdash \top)\) rule is applied (which splits the branch of the proof tree). Finally, the display postulate(s) which were used to display \( A \lor C \) are reversed (in both branches). This last step relies on the fact that the \((\lor \vdash \top)\) rule leaves the term on the right of the ‘\( \vdash \)’ unchanged. Notice that the required display postulates move \( A \lor C \) to the left, so the introduction rule needed is that for introducing \( \lor \) on the left.

\[
\frac{D \vdash \ast A, B}{A \vdash \ast(D, \ast B)} \quad \frac{D \vdash \ast C, B}{C \vdash \ast(D, \ast B)} \quad \frac{D \vdash \ast(D, \ast B)}{A \lor C \vdash \ast(D, \ast B)} \quad \frac{D \vdash \ast(D, \ast B)}{A \vdash \ast(A \lor C), B}
\]

An example of several steps of this form follows. The rules used are the logical introduction rules \((\vdash \rightarrow), (\neg \vdash \top), (\land \vdash \top)\) and \((\neg \vdash \top)\), distribution of \(\ast\) over \(\land\), (DeMorgan’s law), and double negation elimination.

\[
\frac{\ast C \vdash \ast C, A, B}{(\top \ast\ast)} \quad \frac{A \vdash \ast C, A, B}{(\vdash \ast\ast)} \quad \frac{\ast C \vdash \ast C, \neg A, B}{(\neg \vdash)} \quad \frac{A \vdash \ast C, \neg A, B}{(\vdash \neg \vdash \top)} \quad \frac{\ast C \vdash \ast(\neg A, C), B}{(\land \vdash)} \quad \frac{A \vdash \ast(\neg A, C), B}{(\land \vdash)} \quad \frac{C \rightarrow A \vdash \ast(\neg A \land C), B}{(\rightarrow \vdash)}
\]

\[
C \rightarrow A \vdash \ast(\neg A \land C), B \quad (\rightarrow \vdash)
\]

\[
\frac{C \rightarrow A \vdash \ast(\neg A \land C), B}{(\rightarrow \vdash)}
\]

\[
C \rightarrow A \vdash \neg A \land C \rightarrow B
\]
The tactics can be programmed to perform the above steps one at a time, or all in one step; observe that if a proof rule splits a branch of the proof tree, the search-and-rewrite process continues on each branch independently. We now describe how this is implemented in Isabelle.

3.4 Implementation in Isabelle

The implementation of Display Logic in Isabelle is described generally in [6]. Here we describe the implementation of the rewriting procedure. In part, this is conceptually similar to the implementation described for LPF, in that it uses simple functions to build more complex strategies for traversing a term and finding subterms to rewrite; just as described in Sect. 2.3. However, the function that is used to perform a single rewrite is distinctly different from a HOL conversion.

We first introduce the concept of an action, defined by

\[
\text{datatype } \alpha \text{ action} = \text{Unc} | \text{Act of (int } \to \text{ thm } \to \alpha) ;
\]

An action is used to operate on a sequent; Unc means the sequent will be unchanged, whereas Act fn means that fn is used to transform the sequent. For backward proof, fn replaces a subgoal with a number of replacement subgoals; thus when \(sg\) is a subgoal number, \(fn \ sg\) is a tactic (which has type \(\text{thm } \to \text{ thm Seq.seq}\)). For forward proof, we would want \(fn\) to be a function which maps one theorem to another. However, to enable the code to be shared, we let \(fn\) have an extra integer argument which is ignored. Thus for forward or backward proof, the type \(\alpha\) is \(\text{thm}\) or \(\text{thm Seq.seq}\) respectively.

We also have a type \(\text{side}\), whose values are \(\text{Ant}\) and \(\text{Suc}\), denoting the antecedent and succedent (left and right) sides of the \(\vdash\).

Instead of a conversion, we have an action function, of type \(\alpha \text{ actfn = term } \to \text{ side } \to \alpha \text{ action}\). If, for example it is called with arguments term \(t\) and side \(\text{Ant}\), and returns action \(\text{Act act}\), \(act\) should transform a sequent \(t \vdash X\) to another sequent \(t' \vdash X\) (for backward proof, the result may be any number of new subgoal sequents).

Again, we have functions that combine or transform these action functions; these functions differ for forward or backward proof, but to let the forward and backward functions share a lot of the code we define a record type \(\alpha \text{ meth}\), which is described in Appendix A. With this framework, it is easy to construct functions analogous to conversions. We have several operators of which \(\text{THENaf}\), \(\text{REPaaf}\), \(\text{SUBaf}\) are similar to those described in Sect. 2.3, so they are not all described here.

\(\text{THENaf, ELSEaf : 'a meth } \to \text{ 'a actfn } \ast \text{ 'a actfn } \to \text{ 'a actfn}\)

These combine the effects of two action functions in sequence. \(\text{THENaf}\) applies the second whether or not the first succeeds, but \(\text{ELSEaf}\) applies the second only if the first fails.

As in Sect. 2.3 it is easy to create more. For example, \(\text{TDFaf}\), given by
fun TDFaf meth af tm =
    ELSEaf meth (af, SUBaf meth (TDFaf meth af)) tm ;

go through whole term, in top-down order, looking for subterms to change,
but only making the first possible change in any branch of the term’s “structure
tree”. TDFaf was written solely for the purpose of producing the intermediate
steps of the second example in Sect. 3.3.

As explained above, rewriting a subterm involves displaying that subterm,
performing the rewrite action, and then reversing the display step. (The process
of displaying any chosen subterm has been automated, and is described in [6]).
However, in attempting to rewrite a subterm (e.g., where TDAf, which is analogous
to TDTtt, is used to find and perform all possible rewrites) it is important for
efficiency that a subterm be displayed only if the subsequent attempt to rewrite
it will succeed. The coding of SUBaf ensures this; a subterm is displayed only if
it will be changed. This is why the arguments of an action function are a term
t and a side; the action function tests these to see whether a sequent t ⊢ X
or X ⊢ t would be changed, rather than testing the theorem containing the
sequent. Once a subterm is identified to be changed, though, we may need to
test the new subterm for further possible changes; in this case we have to wait
until the original subterm is displayed and changed before we can ascertain the
new subterm.

We have to programme the action functions. This is achieved for forward
proof as follows. Given a term t, on (say) the antecedent side, a rule rule of the
form A ⊢ X ⊑ A' ⊢ X, which instantiates to t ⊢ X ⊑ t' ⊢ X, is used; then
Act f, where f takes t ⊢ X to t' ⊢ X, is returned. Typically rule is selected from
a specified set of candidate rules.

For backward proof, the procedure is similar, except that the rule rule may
have several premises, of the form

||[B_1 ⊢ X; \ldots; B_n ⊢ X]| ⊢ A ⊢ X

and the action returned takes the goal A ⊢ X (instantiated) to the several
subgoals B_1 ⊢ X, \ldots, B_n ⊢ X (instantiated).

4 Reflection: Why is it possible?

We now try to elucidate the aspects of these logics which make the subterm
rewrite capabilities possible.

The technique for rewriting in LPF can be explained and justified in terms
of Window Inference. In Window Inference, as described by Robinson & Stan-
ple [20], one could transform an expression such as (A → B) ∧ C to an equiva-
 lent expression by focusing on A, and transforming it to an expression A', where,
under the assumptions ¬B and C, A is equivalent to A'. Grundy [14] shows how
this could be extended from equivalence to any pre-order, and discusses the ex-
tension to non-classical logics; he has also implemented it (for classical logic) in
HOL [15].
We can fit the results on LPF into the Window Inference framework. The following results are in the format of window rules [14, Sect. 3.4].

\[
\begin{array}{c}
x \sqsubseteq y \\
g(x, \ldots) \sqsubseteq g(y, \ldots)
\end{array} \quad \begin{array}{c}
p \sqsubseteq q \\
p \implies q
\end{array}
\]

where \( g \) is as in Lemma 1 and \( p \) and \( q \) are boolean. These two results combine to give Lemma 1. Window Inference using these rules is equivalent to the procedure we have described. It should be noted that these window rules do not introduce any contextual assumptions.

It seems much more difficult to formulate, in terms of Window Inference, the method we will describe for rewriting in Display Logic. This is because, in Display Logic, although we “focus” on a sub-expression, the logical steps used always involve the whole expression (usually a variant of it obtained using the display postulates, but an expression of roughly the same size).

The following discussion may help to indicate the underlying differences between the methods.

Consider a group (of which an example would be real invertible \( n \times n \) matrices), with the multiplication operation ‘\(*\)’. Given an equation \( A = B * C \), we can “display” \( B \) by rewriting the equation as \( A * C^{-1} = B \), making use of the invertibility of \( C \). Suppose we also have an equality (which may be a general rule instantiated), \( B = D \). Then we can use the transitivity of equality to replace \( B \) by \( D \), and reverse the transformation by which we displayed \( B \), thus:

\[
\begin{align*}
A &= B * C \\
A * C^{-1} &= B \\
B &= D \\
A &= D * C
\end{align*}
\]

Now consider a semigroup (of which an example would be all real \( n \times n \) matrices, with the multiplication operation). Here we cannot take the inverse of a given element. However, given the equality \( B = D \), we can multiply both sides by \( C \) to get \( B * C = D * C \), and prove the same result thus:

\[
\begin{align*}
B &= D \\
A &= B * C \\
B * C &= D * C
\end{align*}
\]

This second proof uses the congruence of equality, in the form

\[
\begin{align*}
B &= D \\
C &= C' \\
B * C &= D * C'
\end{align*}
\]

In that way the method used strongly resembles the method used for LPF, where there is repeated use of rules of the form

\[
\frac{\text{rep}(B,D)}{\text{rep}(B \text{ op } C, D \text{ op } C')}
\]
The method of the first proof above has some similarity to the method used for Display Logic, in that a selected subterm is “displayed” and transformed, and then the steps used to “display” the subterm are reversed. However, Display Logic does not have equalities or inverses; the ability to display a chosen subterm relies on the other features of the underlying logics that enable residuation, as discussed in Sect. 3.2.

5 Conclusions and Further Work

Methods of imitating the capability to rewrite subterms have been presented for LPF and Display Logic. Since Display Logic does not have equality of subterms, and LPF does not have equality of the possibly undefined terms used, in neither case does the logic explicitly support such rewrites. However, in each case, certain features of the logic enable the effect of a general rewrite capability to be achieved; tactics have been written to exploit these features of the logics, to make the “rewriting” fairly straightforward for the user.

Of the methods described in this paper, that for Display Logic has been used extensively, and proved very valuable in facilitating a number of long proofs. Examples of these are given in [6]. This was no surprise; given the prevalence of term rewriting as a proof method, it is to be expected that a method achieving the same effect would be valuable. The method has been adapted to several different display logics. Recent re-implementation work has made it much easier to implement a new display logic in Isabelle, and then the tactics described in Sect. 3 will apply automatically. The tactics are fairly straightforward to use.

In the case of the method for LPF, so far as the user is concerned, the tactics are reasonably easy to apply. However, they have been implemented so far only for classical first-order logic, and have not yet been used significantly. Extension to other theories would require further work; in particular, it would be necessary to derive rules of the form of (1) for every monotone operator in the system. Some other programming (eg adapting the function which selects the appropriate rule of the form (1) at every step, and adapting some functions to handle multiple Isabelle types) would also be necessary.

Both methods have been implemented so as to permit the user to specify the strategy for traversing a term looking for subterms which can be rewritten, as can be done with HOL conversionals. So far the “ordered” rewriting which Isabelle’s rewrite tactics use for “permutative” rewrite rules (see [18], Sect. 10.4) has not been implemented; this potential further work would not be difficult.

Reade ([19], Ch. 10) describes various constructions for domains. Briefly, a flat domain has only the usual elements of the type, plus an undefined element $\bot$. Alternatively, a domain can have “partly undefined” elements, such as a tuple $(3, \bot)$. This means that there would be a choice of ways of extending LPF to handle compound types. Possible further work in this area would include analysing the implications of such choices on the use of the methods described in Sect. 2.
Acknowledgement

The author wishes to thank Rajeev Goré and Peter Lindsay for helpful comments and discussions, and unnamed referees for useful suggestions.

References

A Isabelle code for Display Logic Rewriting

The record type \( \alpha \texttt{meth} \) has the following fields (note that here the arguments referred to as "actions" do not have the \texttt{Act} prefix):

\[
\begin{align*}
\text{idact} & : \texttt{int -> thm -> } \alpha \text{ is the identity action, the action which doesn't change the sequent} \\
\text{actcomb} & : \ (\texttt{int -> thm -> } \alpha) \ast (\texttt{int -> thm -> } \alpha) \rightarrow (\texttt{int -> thm -> } \alpha) \\
\text{dpact} & : \texttt{thm -> int -> thm -> } \alpha \text{ gives the action to be taken to transform a term using a given display postulate} \\
\text{find_t} & : \texttt{int -> thm -> term} \text{ is the function which finds the term being examined, i.e., in backwards proof, the subgoal, or, in forward proof, the conclusion of the theorem.}
\end{align*}
\]

The source code defining the values used for these arguments is

\[
\begin{align*}
\text{for backward proof} & \\
\text{fun idact sg = all_tac ;} \\
\text{val actcomb = op THENEXP ;} \\
\text{val dpact = rtac o md2 ;} \\
\text{fun find_t sg state = nth (prems_of state, sg-1) ;}
\end{align*}
\]

\[
\begin{align*}
\text{for forward proof} & \\
\text{fun idact _ th = th ;} \\
\text{fun actcomb (f, g) _ = (g 0) o (f 0) ;} \\
\text{fun dpact theq _ th = th RS md1 theq ;} \\
\text{fun find_t _ _ = concl_of ;}
\end{align*}
\]

In the above, \texttt{md1} and \texttt{md2} turn a meta-equivalence to forwards and backwards meta-implications; \((\texttt{tacf1 THENEXP tacf2}) sg\) applies \texttt{tacf1} to subgoal number \(sg\), then applies \texttt{tacf2} to each resulting subgoal. This tactical may be generally useful; it corresponds to the HOL tactical \texttt{THEN} ([9], Sect. 14.4.4), and to the tactical \texttt{THEN_ALL_NEW_SUBGOALS} as described by Easthaughfe et al in [8], p. 378.

A Prototype Generic Tool
Supporting the Embedding
of Formal Notations*

Andrew M. Gravell and Chris H. Pratten
University of Southampton, Southampton SO17 1BJ, England
gravell@soton.ac.uk

Abstract. In this paper, we describe the design and implementation of a
prototype tool designed to support the embedding of one formal notation
within another. The tool is designed primarily to support the automatic
embedding of specification notations such as Z or AMN into the notations
of generic theorem provers such as HOL [1] or PVS [2]. It is written
in Java [3], which enables novel features such as dynamic extensibility.
There is a common intermediate form comprising a collection of classes
called LIL (Logic Interface Language) permitting the clean separation of
parsing (specification notation) from printing (theorem prover notation).

1 Introduction

Tool support for specification notations such as Z [4], VDM [5] and AMN [6]
has often been weak especially in the area of proof. Building a theorem prover is
however a quite specialist task requiring a lot of effort particularly in the areas of
building sophisticated tactics and theorem libraries. There has been some interest
of late in using general purpose (often freely available) theorem provers for
proving properties of the above specification notations [7,8]. Most of the literature
however concentrates only upon demonstrating, by manual translation, the
possibility of an embedding. In this paper we describe the prototype JavaLIL tool
supporting the automatic embedding of statements in formal notations such as Z
and AMN into other formal notations and in particular the notations of generic
theorem provers such as HOL and PVS. We discuss the design issues involved
in the construction and our experiences of constructing a Java implementation.

The two noteworthy features of JavaLIL are included in its name. Firstly it is
implemented in Java, which enables novel features such as dynamic extensibility.
Secondly it makes use of a common intermediate form comprising a collection of
classes called LIL (Logic Interface Language). This permits the clean separation of
parsing (specification notation) from printing (theorem prover notation). This paper
concentrates in particular on these two aspects of the tool.

An alternative approach is represented by ICL’s ProofPower tool [9], which is
a version of HOL re-implemented to support a version of Z as its logical language.

* The tool was constructed as part of the EPSRC project *Tools Integration for Applied
  Formal Methods* (grant GR/K83014).
This approach has the advantage that users do not need to know HOL’s original logic, though of course they must still learn how to use the prover effectively; this may take several months compared with, say, a couple of weeks to learn a new notation [10]. Our approach enables developers to choose which prover they wish to apply. It may be that one developer has more experience of HOL and another has more experience of PVS. Moreover, thanks to the common intermediate form, our tool has been cheaper to develop, and can more easily be extended to support other formal notations. The situation is similar to that in programming language compilation where use of a common intermediate form allows more languages to be supported more cheaply, but individually crafted translators usually produce higher quality translation. Similar remarks can be made about natural language translation.

In Sect. 2 we describe the architecture of the tool. In Sect. 3 we consider its implementation. Section 4 concentrates on our experiences of Java as an implementation language for this system, and in Sect. 5 we examine in more detail the problems of providing an extensible abstract syntax. We then end with some conclusions and a consideration of some possible extensions and improvements to this work.

2 Architecture

The following diagram illustrates the overall architecture of the JavaLIL tool.

![JavaLIL architecture diagram](image)

**Fig. 1.** JavaLIL architecture diagram

Supporting a target notation consists of providing a mapping from elements of basic LIL (and possibly from some source specific specialisations) to a textual
form appropriate to the target solution. This is achieved by writing a new parser and adding a pretty printer configuration as discussed in the following section.

We chose Java [3] as our implementation language because of its portability and its support for extensibility. An object oriented language gives us the best facilities for constructing a securely extensible tool core. We want it to be as easy as possible for a user to add new components to the system. In particular the addition of new parsing components should not compromise existing components and if possible should not compromise any existing persistent binary data. It is desirable that extensions may be distributed in binary rather than source code format so that the user does not need to run a compiler (or run them more slowly via an interpreter). Java gives us all this and more. With Java, unlike C++ [11] we may add components dynamically to a running system.

Our tool architecture is designed as a central core of Java classes implementing support for what we originally termed a Logical Interface Language (LIL). However there are doubts that such language is appropriate [12]. And indeed in the final design this interface is purely internal, so that LIL may be thought of as defining a simple abstract syntax: we do not provide any concrete representation of the interface logic. Since the interface logic is defined as a class, or rather a collection of classes, LIL is in fact more than just syntax. Its constructors incorporate syntax- and type-checking and there are also methods for substitution and renaming of variables, which is required for the instantiation of parameterised modules.

A source notation is supported by extending, or forming sub-classes of, the basic LIL classes, and implementing a parser. The LIL classes are constructed so that any extensions preserve type correctness and other consistency requirements. For example, LIL has a class for constructing lambda abstractions and another for constructing applications. The application constructor ensures that only a type safe application may be formed. The attributes of an application are also fixed to ensure that an initially well-formed application cannot be changed into an ill-formed one by changing the value of, for example, the rator attribute.

A source notation requiring a construct such as, for example, a let expression might define and use a sub-class of the applications. 1 The let expression constructor then controls the expression semantics by how it forms the underlying application. Extension by sub-typing is particularly powerful in this context.

If the LIL classes have been extended it may be desirable to add pretty-printer configurations that tell JavaLIL how to print objects of the sub-class, rather than inheriting this from the basic LIL class.

3 Implementation

The core of the JavaLIL tool is formed by the Java classes in the package JavaLIL.BasicLIL. All formal source notation constructs are supported by these classes or an extension of them. The classes implement terms and types of higher

---

1 PVS supports let expressions in this way.
order logic. These are constructed with respect to a module which defines the context of the term or type. A module is the basic container for definitions and declarations in LIL. In addition to type and term parameters, the module parameters may also contain another imported module. This allows, for example, the type of a module term parameter to depend on type constants imported from another module. Modules can either be named, forming their own context, or else anonymous. Anonymous modules are used for generic definitions which introduce extra (term or type) parameters in a block-structured way into a containing module.

The parser takes elements of source for the given notation and constructs basic LIL elements either directly or by constructing some specialisation. In theory an embedding could be made entirely into basic LIL elements, but this might in practice mean losing a lot of information about the original elements. Extensions to the basic LIL classes are also a convenient mechanism for structuring any notation specific code such as extra type-checking.

We have constructed prototype embeddings for the Z and AMN notations. The parsers are constructed using the JavaCC tool from published grammars. Each embedding involves a number of extensions to the LIL classes at all levels from terms and types to modules. For example we define the class of lets expressions as a specialisation of the basic LIL application. Z generic constant definitions are defined as a special class of anonymous module which has constant definitions for each Z generic constant and is parameterised by the Z generic type parameter.

Each target notation is supported by adding a pretty printer configuration which maps basic LIL (and any source specific specialisations) to an appropriate textual form.

The pretty printer configuration is a mapping from Java classes to guarded formatters for the classes. A class formatter is simply an object describing how to map an object of the class to a textual representation. We have defined a simple language (based on the Java syntax) for describing the configuration settings and implemented a parser to allow the pretty printer to read configuration scripts.

To give an example, the basic LIL class representing applications, might be given the following formatter definition.

```java
JavaLIL.BasicLIL.TermApplication *
{
    this.getRator();
    "(n;
    this.getRand();
    ")n;
}
```

This states that the class `JavaLIL.BasicLIL.TermApplication` should have a default (un-guarded) formatter which first pretty prints (using the current setting) the rator of the application, then next prints an opening parenthesis, then pretty prints the rand of the application and finally prints a closing parenthesis.
Given an object to pretty print, the pretty printer first attempts to use a
guarded formatter for the class, then tries to use the default formatter, if this
fails or none are defined the search proceeds using the super-class of the object
class.

The pretty printer makes considerable use of Java’s reflection facilities. These
allow us to type check and partially compile a pretty printer configuration
script. In processing the example above the pretty printer checks that the class
JavaLIL.BasicLIL.TermApplication does indeed have methods getRand and
getRow, and obtains objects representing these methods. This removes the
inefficiency of looking up the methods each time the pretty printer is evaluated
on an object of the class (hereafter known more simply as TermApplication).

We have designed JavaLIL to be, as far as possible, open and extensible. The
system consists essentially of a number of components implementing parsers,
pretty printers, basic LIL classes and extensions.

To demonstrate the functionality of the tools we have constructed three user
interfaces using Javascript running under Netscape Navigator, Java AWT and
Java Swing. Each of these allows the user to dynamically load and configure
parsers, pretty printers or LIL extensions and then use their standard facilities
to load, check and print Z and AMN specifications. Thus, for example, once
a Z module has been loaded, parsed and type-checked, different pretty printer
configurations can be selected and used to generate HOL, PVS, or a EMiX
representation, without needing to reload and reparse the Z.

We have used three specification case studies to validate the tool

- Business Applications Manager: we are grateful to IBM Hursley for allowing
us access to this AMN specification.
- Smart Card Object Manager, a Z specification developed in collaboration
with Integrity Arts.

These specifications all parse and type-check correctly. The standard proof obligations
for soundness of specification [6,14] and refinement [4] were generated in
both HOL and PVS. A sample of these proof obligations were discharged using
these theorem provers to convince ourselves that the embeddings are correct.

4 Use of Java

Tools supporting embeddings have been written before [8,9,14]. The novel fea-
tures of JavaLIL, such as its extensibility and configurability, follow from its
implementation in Java. This section discusses advantages, disadvantages and
possibilities for further use of Java’s features.

We have been impressed by the quality of Java’s libraries, specifically the
collection classes and the support for windowing and events; the tools, including
JavaCC, command line and IDE compilers; and language and execution features
such as inheritance, dynamic linking, and reflection including run-time type identification. The modest development effort involved (just over a person year) is tribute to the effectiveness of Java as a development tool.

The portability of Java is another bonus. The languages we are supporting are supported by a variety of tools often running on disjoint sets of platforms. For example, Nitpick [15] (a Z model checker) currently only runs on a Macintosh, PVS only runs on Unix platforms and it is likely that many new tools and users will use Windows 95 or NT.

Our implementation makes use of the final declaration to protect attributes and methods against modification even if the base LIL classes are extended using inheritance. This means that new code cannot, for example, compromise type-checking. In this way the flexibility of object-oriented programming can be combined with the security of a traditional abstract data type. This issue is discussed in more detail in the following section.

Java does of course have its disadvantages. Fortunately, most of these are due to its immaturity and are not intrinsic. We have found that the performance of Java, even with Just In Time compilation, is less than adequate, up to an order of magnitude worse than that of a similar tool we constructed in C++ [14]. Memory utilisation is less efficient also. The language stability and the robustness of Java systems for both execution and development has also been a concern.

We have on a number of occasions found ourselves swapping between compilers or interpreters to find a working combination. However it would seem that with the arrival of JDK 1.2 the language and its support tools are starting to mature and stabilise.

5 Providing an Extensible Abstract Syntax

We have found a weakness in our implementation arising from our use of inheritance. Renaming or substitution often unnecessarily reduces the term representation to that of the basic LIL class. To understand how this arises it is necessary to consider details of the classes involved.

The abstract class Term has a single package private constructor. This ensures that only classes within the same package can inherit from Term. There are three sub-classes of Term: variables, TermVariable, lambda- abstractions, TermAbstraction, and applications, TermApplication. The most important attributes of a TermApplication are its operator rator and operand rand. These are supplied by the constructor TermApplication(Term Rator, Term Rand) which checks their types and throws an exception if there is a type error. The rator and rand attributes are declared final private, and their accessor methods getRator and getRand are final public.

Now, the tt rator and rand cannot be changed, the constructor includes type-checking, and moreover Java guarantees that this constructor is called first whenever any new instance is created of this class, or any derived sub-class. It is therefore certain that all TermApplications are well-typed.
All Terms provide a method to replace a variable with a term. This is used, for example, to instantiate module level parameters. The code for the substitution method in the TermApplication class is essentially as follows

```java
final public Term substitute(TermVariable v, Term t) throws TypeException
{
    final Term newRator = rator.substitute(v, t);
    final Term newRand = rand.substitute(v, t);
    return (newRator == rator && newRand == rand)
        ? this
        : new TermApplication(newRator, newRand);
}
```

Unfortunately the constructor invoked in the final line is the one belonging to the base TermApplication class. This means that a substitution will convert any kind of application to a basic one. As a concrete example, consider let expressions which are implemented in JavaLIL (and indeed in HOL) as a special kind of application. The user, however, does not wish to see let \( x = e \text{ in } t \) converted to \( (\lambda x.t)(e) \) just because the let expression happens to appear in an imported module. To avoid this “down-conversion”, at least in trivial cases, a test is applied to see if neither of the terms has changed, in which case the original application is returned unchanged. Unfortunately, this is only a partial solution to the down-conversion problem.

What we need is a dynamically dispatched method call, not a static constructor. The standard solution to this problem involves a design pattern known as a factory method [16]. A makeApplication method is added to the basic LIL TermApplication class which has the same parameters and behaviour as the constructor, and this method is called by substitute instead of the constructor.

```java
public Term makeApplication(Term Rator, Term Rand)
{
    return new TermApplication(Rator, Rand);
}
```

```java
final public Term substitute(TermVariable v, Term t)
{
    final Term newRator = rator.substitute(v, t);
    final Term newRand = rand.substitute(v, t);
    // use the factory method rather than the static constructor
    return makeApplication(newRator, newRand);
}
```

Each subclass can provide its own version of makeApplication since it is not declared final. This is simpler, and safer, than having each sub-class make its own implementation of substitute.
An alternative solution to the problem uses reflection to obtain access at run-time to the desired constructor:

```java
final public Term substitute(TermVariable v,Term t)
   // exception handling code omitted for brevity
   {
      final Term newRator = rator.substitute(v,t);
      final Term newRand = rand.substitute(v,t);
      // use the reflection API to invoke the correct constructor
      Class thisClass = this.getClass();
      Constructor cons = (new Class this.getClass()).getConstructor(
         new Class[] {Class.forName("Term"),Class.forName("Term")});
      return (TermApplication)cons.newInstance(
         new object[] {newRator,newRand});
   }
```

This requires more code in the class TermApplication but none in its subclasses: it is less readable but makes extensions less error-prone. We are at present unsure which approach to adopt. The use of Java and Design Patterns in theorem proving tools is discussed further in a companion paper [17].

6 Further Work and Conclusions

The JavaLIL tool currently provides some useful and interesting functionality but more work is required before the translations can be of practical use. For example, more source-specific lemmas and tactics would need to be constructed for the target provers. The target embeddings would also benefit from additional configurations specific to the source notations to improve their readability.

An important area of potential improvement is in making the tools translation more bi-directional. At present we have concentrated on ensuring a natural relationship between the translated form and the original. It would be more satisfactory if we could make the JavaLIL tool provide an automatic translation back from target elements to the original, particularly if this reverse translation could work on a wider range of target terms than those directly generated in a translation.

We have found the level of direct integration possible with existing provers quite disappointing. Only HOL is sufficiently open to allow any sort of interaction more sophisticated than a text stream. We have experimented with only limited success with providing a CORBA interface to HOL. Neither of the provers provided any sort of useful configuration mechanisms, for example, to alter or extend the language of the prover, which would be an alternative way to provide JavaLIL's functionality. Isabelle is a rare example of an extensible proof system [19].

2 Another companion paper discusses this aspect of the translation [18].
In this paper we have provided an overview of the JavaLIL tool. The tool is designed as an extensible and configurable tool supporting the embedding of formal notations. It is implemented in Java, which has been successful. We have discussed the advantages of Java for this application. Experimental source notation support has been constructed for both Z and AMN. Pretty printer configurations have also been constructed for outputting to HOL, \LaTeX{} and PVS.

References

18. A. M. Gravell and C. P. Pratten. Embedding a formal notation: Experiences of automating the embedding of Z into the higher order logics of PVS and HOL. University of Southampton, 1998.

Embedding a Formal Notation: 
Experiences of Automating the 
Embedding of Z in the Higher Order 
Logics of PVS and HOL*

Andrew M. Gravell and Chris H. Pratten
University of Southampton, Southampton SO17 1BJ, England
gravell@soton.ac.uk

Abstract. In this paper, we consider the problem of embedding formal notations. In particular, we describe our experiences of automating the embedding of Z specifications into the notations of the PVS and HOL theorem provers. This paper is motivated by our experiences of constructing a prototype tool for embedding formal notations and its use in automating an embedding of Z and AMN into the notations of PVS and HOL.

1 Introduction

Tool support for specification notations such as Z [1], VDM [2] and AMN [3] has often been weak especially in the area of proof. Building a theorem prover is a quite specialist task requiring a lot of effort particularly in the areas of building sophisticated tactics and theorem libraries. There has been some interest of late in using general purpose (often freely available) theorem provers for proving properties of the above specification notations [4,5,6,7,8]. Most of the literature however concentrates only upon demonstrating by example the possibility of an embedding. In this paper we describe our experiences of supporting a fully automatic embedding of Z into PVS [9] and HOL [10].

Two noteworthy features of the tool are included in its name: JavaLIL. Firstly it is implemented in Java, which enables the tool to support dynamic extensibility. Secondly it makes use of a common intermediate form called LIL (Logic Interface Language) which is implemented as a collection of classes. This permits the clean separation of the front end of the tool from the back end, so that, for example, two source notations (Z and AMN) are supported, as well as two output notations (PVS and HOL). A companion paper [11] concentrates in particular on these aspects of the tool.

The focus of this paper is the nature of the supported embeddings.

An alternative approach is represented by ICL’s ProofPower tool [12], which is a re-implementation HOL with Z as its logical language. This approach has the

* This work has been funded by the EPSRC (grant GR/K83014).
advantage that users do not need to know HOL's original logic, though of course they must still learn how to use the prover effectively; this may take several months compared with, say, a couple of weeks to learn a new notation [13]. Our approach enables developers to choose which prover they wish to apply: it may be that one developer has more experience of HOL and another has more experience of PVS. Moreover, thanks to the common intermediate form, our embedding tool has been cheaper to develop, and can more easily be extended to support other formal notations.

Many of the problems associated with this automatic translation are common to other forms of translation such as programming language compilation. But there is a very important difference between the translations we are interested in and those performed by, for example, a compiler. We require that the resultant embedding must be readable by a user and as far as possible understandable in terms of the original specification. We illustrate this in the following diagram.

The mapping back from embedded form does not need to be automatic, unique or even always exist, but the user should have some intuition of the relationship with the original notation.

Note that we are considering here only the problems associated with reasoning about specifications written in Z and not about statements about the Z language itself. Formal semantics, such as Spivey's [14], can be expressed as an embedding, but would lead to more complicated and less readable expressions in the target logic. For example, a full semantics must model Z's generic parameterisation mechanism rather than simply making use of equivalent features in the target system.

1.1 The Z Notation

The Z notation [1] is a system specification notation based on classical set theory, but with the addition of types.

The Z syntax is particularly flexible. The Z standard allows user definition of infix and post- or prefix operators as well as a very liberal definition of allowed identifiers. The full standard syntax is also highly context dependent, even to the extent that white space is in certain circumstances interpreted as predicate conjunction.
Z specifications are structured using a construct known as a schema. The Z notation contains a rich set of constructs for composing schemas giving a user a very powerful notation for structuring system specifications. Semantically a schema may be considered to define a set of bindings. For example the following schema defines a set of two element bindings where the binding denoted by first is always strictly less than that denoted by second.

\[
\text{OPair} \\
\text{first : Z} \\
\text{second : Z} \\
\text{first < second}
\]

The draft standard [15]\(^1\) also allows implicit instantiation of generic type parameters. For example, when using a generic list the list type does not need to be explicitly stated if it can be inferred from the context.

### 1.2 HOL and PVS

HOL [10] and PVS [9] illustrate the current state of the art in semi-automatic theorem provers. By semi-automatic we mean a prover that combines support for both automatic proof progress where possible with support for driving the proof process interactively when necessary.

Both these provers support a typed higher order logic. There are however some quite important differences in the notations supported. PVS supports a richer notation in particular supporting dependent types, automatic type conversions and parameterised theories.\(^2\) In fact the only area where the HOL notation is not essentially a strict subset of that of PVS is type parameterisation. HOL has a much richer support for parametric types. Any HOL term or definition may be dependent on a set of type variables that may be instantiated when the term or definition is used. This means that in HOL type parameterisation is at the level of individual definitions. This is contrasted by the PVS mechanism where type parameterisation can only occur at the theory boundary.

### 2 Preserving Semantics in a Mapping

Clearly a useful automatic embedding must preserve the semantics of the original specification. The embedding of typed set theory in HOL or PVS is quite well understood [16] so we really need consider here only the extensions of Z to typed set theory.

The main distinctions of Z are the mechanisms for declaring and constructing schemas. We will concern ourselves in this section mostly with the problem of embedding the schema calculus in HOL and PVS.

---

\(^1\) See also the 1995 draft http://www.comlab.ox.ac.uk/oucl/groups/zstandards/

\(^2\) A PVS theory is simply a parameterised container for definitions and theorems.
Lets begin by looking at possible PVS and HOL representations of the schema OPair defined above. The semantics of the schema should be a set of bindings. In higher order logic sets are represented as functions with codomain the Booleans. So we require a function from the two element binding to the Booleans. The PVS representation might be as follows

```
OPair (s:[# first: int, second: int #]) :
  bool =
  let
    first = first(s),
    second = second(s)
  in
    first < second
  end
```

This is a not too unnatural representation of the original schema. Unfortunately, as we are considering here an automatic translation, in general the above representation would be a little less clear. The constraints on the elements of the binding may be that they belong to an arbitrary set. An automatic translation should support this more general case. The more generally applicable representation would require the addition of explicit constraints on each binding elements to the appropriate declared set. The representation would therefore look something like the following.

```
OPair (s:[# first: int, second: int #]) :
  bool =
  Zint(first(s)) &
  Zint(second(s)) &
  let
    first = first(s),
    second = second(s)
  in
    first < second
  end
```

where Zint is a constant defined to be the full set of integers.

In the case of HOL we have more of a problem. HOL does not have native support for bindings (records) we are therefore forced to find an alternate representation for a binding.\(^{5}\) This problem has been considered before. In the Refinement Calculator [17] records are represented as tuples and a let expression is used to maintain a record of the binding names in the syntax. For example the PVS record (# first := 1, second := 2 #) would be represented as the HOL term

\(^{5}\) An alternative would be to extend HOL to support records: unfortunately such extensions must be compiled into the core system.
LET first = 1, second = 2 IN
(first, second)

A disadvantage of this approach is that semantically inequivalent Z terms are
mapped to equivalent HOL terms. For example the terms (# a := 1, b :=
2 #) and (# b := 1, c := 2 #) result in equivalent HOL terms (after α-
conversion) but are not equivalent Z terms. This will not however compromise
consistency. The reason is that we are identifying terms with differing Z types
and these terms are incomparable in Z.

We'll now proceed to consider the embedding of more complicated schema
calculus constructs. A very common schema expression is one using the schema
logical operators. We'll consider here the case of schema conjunction. The fol-
lowing Z text defines a new schema OPair2 together with a definition of the
schema OTriple as the conjunction of Opair and Opair2.

\[
\begin{align*}
OPair2 \\
\text{second} : \mathbb{Z} \\
\text{third} : \mathbb{Z} \\
\text{second} < \text{third}
\end{align*}
\]

\[
OTriple == OPair \land \text{Opair2}
\]

The definition of OTriple is equivalent to

\[
\begin{align*}
\text{OTriple} \\
\text{first} : \mathbb{Z} \\
\text{second} : \mathbb{Z} \\
\text{third} : \mathbb{Z} \\
\text{first} < \text{second} < \text{third}
\end{align*}
\]

The PVS representation is complicated by the fact that we do not have structural
sub-typing in PVS. That is, for example, the expression (# a := 1, b := 2,
c := 3 #) cannot be used as a value of type [# a: int, b: int #]. We need
to explicitly promote between the record types. A possible definition of OTriple
might be as follows.

\[
\text{OTriple} : [[\text{# first: int, second: int, third: int #} \rightarrow \text{bool}] = \\
\Lambda s:\text{[# first: int, second: int, third: int #]} : \\
\text{LET} \\
\text{first} = \text{first}(s), \\
\text{second} = \text{second}(s), \\
\text{third} = \text{third}(s) \\
\text{IN} \\
\text{OPair} ((\text{# first := first, second := second #}) \land \\
\text{OPair2} ((\text{#second := second, third = third #}))
\text{END}
\]

This is clearly a much less compact representation than the original but it is not hard to see the relationship to the original. The HOL representation is similar but complicated by the representation of bindings as tuples.

A second area with potential problems is the treatment of generics. Z allows both generic definitions and generic schema definitions. Thus we require the ability to make definitions parameterised by type. This is of course not a problem in HOL where type parameterisation of definitions is well supported. However, in PVS type parameterisation is only allowed at the theory boundary. Hence given a set of generic definitions we need to split the definitions between a number of PVS theories. Consider for example the following generic schemas.

\[
\begin{align*}
S[X] & \quad \cdots \\
T[X] & \quad \cdots \\
U[X] & \quad \cdots
\end{align*}
\]

In PVS we need to split these amongst three separate theories, for example.

```
Definitions__1 [X:TYPE] : THEORY
BEGIN
  S = ...
END Definitions__1

Definitions__2 [X:TYPE] : THEORY
BEGIN
  IMPORTING Definitions__1
  T = ...
END Definitions__2

Definitions__3 [X:TYPE] : THEORY
BEGIN
  IMPORTING Definitions__2
  U = ...
END Definitions__3
Definitions : THEORY
BEGIN
  IMPORTING Definitions__1
  IMPORTING Definitions__2
  IMPORTING Definitions__3
END Definitions
```
Note that the split is necessary even though the type parameter is the same in each case. This is because, for example, $S$ may potentially be used in the definition of $T$ with any instantiation of the generic parameter $X$. Although the problem of representing $Z$ generics is solvable in PVS, the need to split in this way between a number of theories certainly compromises readability. In HOL we can avoid this problem because of the wider support for type parameterisation of individual definitions, which fits more closely with the $Z$ model.

Another area of potential conflict in the embedding is the treatment of partial functions and undefinedness. The $Z$ standard is rather vague on this point simply enumerating a number of possibilities. These are essentially

1. A partial function applied to a value outside its domain takes some value in the codomain type.
2. A partial function applied to a value outside its domain is not considered well typed.

We have chosen the first option in our embedding. Recall that in the $Z$ notation [1] all functions are modelled as relations, sets of pairs, together with an additional constraint, usually implicit in the declaration, indicating whether the function is total, partial, injective, and so on. In particular $f \cup g$ is valid $Z$ whenever $f$ and $g$ are functions of the same type. It is therefore in general necessary to translate the $Z$ function $f : A \rightarrow B$ into a PVS or HOL relation, that is $f : [A,B] \rightarrow \text{bool}$. We also model the $Z$ application $f(x)$ by $\text{apply}(f,x)$ where

$$\text{apply} : \text{[]} [A,B] \rightarrow \text{bool} \rightarrow A \rightarrow B$$

satisfies,

$$\text{apply} \ f \ x = \varepsilon(\lambda y : B. \ f(x,y))$$

This uses the choice function $\varepsilon$ (epsilon) which, given a predicate of type $\tau$, produces an element of $\tau$ satisfying that predicate if one exists and otherwise produces an arbitrary value of that type. (In HOL one writes $\emptyset : \tau$ rather than epsilon.)

It is worth noting that should we have chosen option 2 for the semantics of application, then the representation would have been much easier in PVS. In HOL, we would have had to construct an external proof obligation generator to generate the additional theorem necessary to show that partial function application was everywhere well typed. In PVS we may simply use the dependent type system to check that all function applications are well typed. The definition of $\text{apply}$ in PVS would simply require the new dependent type

$$\text{apply} : (f : \text{ZParFun}[A,B]) \rightarrow (\text{dom} \ f) \rightarrow B$$

In a study of compiler correctness [8], it is recommended that exact domains are given for each partial function. This is probably good practice for a manual embedding, but it is not in general possible for an automatic one.

\[ \text{ZParFun}[A, B] \] is the expression representing the set of partial functions with domain $A$ and codomain $B$. \[ \text{ZParFun}[A, B] \]
3 Mapping Syntax

In the previous section we considered the main difficulties in providing a Z embedding that preserves the semantics. If we just wish to automatically translate a Z specification but never look at the results then we could stop here (this might be the case if we wanted to perform some automated checking of the code, for example model checking). Unfortunately we need to be able to perform meaningful interactive proof on the translated specifications. As discussed in the introduction, this requires that there is an intuition of the mapping from the translation back to the original.

The problem can be subdivided between what we will term micro and macro level constructs. By micro level constructs we mean the low level term representation. By macro level we mean the overall specification structuring constructs: how the specification may be sub-divided between a number of theories. These levels are obviously not entirely independent.

We'll begin by looking at macro level constructs and work towards the micro level. Fortunately Z, as defined in the standard, does not have any real structuring constructs above the schema level.\(^5\) In our embedding we support a crude notion of module for grouping sets of schemas and definitions. In the case of HOL, because of the richer type parameterisation mechanism, we are free to map our Z modules directly to similarly named HOL modules. In PVS as outlined in the preceding section the less rich type parameterisation means that we must assign each generic definition to a separate module. This means a set of schema definitions might be split amongst a number of PVS theories thus obscuring the intuition of the reverse mapping.

Having distributed the schema definitions to appropriate theories (or a single theory for HOL) we come now to the representation of individual schema or constant definitions. For simple schema or definitions we can just use the constant definition mechanisms of the host notation. The representation, although not Z in style, is not hard to understand in terms of the original Z. For generic schemas, axiomatisations and particularly generic constants the representation is less tidy. Let's consider the general case of the simultaneous definition of a number of generic constants.

\[
\begin{array}{l}
[X] \\
\quad \text{first} : X \\
\quad \text{second} : X \\
\quad \text{third} : X \\
\quad \ldots
\end{array}
\]

We have a number of options in translating this construct. We clearly need constant definitions for each of the generic constants first, second and third. We do not however want to repeat the predicate defining the constant properties in

\(^5\) There are however various proposals for adding additional structuring constructs [18].
Each case. This would be very non-intuitive and would obscure any relationship between the constants. We really want to define a local definition of a constant representing the predicate and then definitions for each of the constants. Unfortunately neither HOL nor PVS provides good support for name hiding. We are forced therefore to generate a name for the constant, which does not clash with any other. We then make this constant represent the generic schema defined by the constant declaration. The generic constants are then defined to form a binding satisfying the schema. In PVS, we must first parameterise the enclosing theory by the type X. But as we also want to use X as an expression we rename the type to X-t. The PVS generic schema definition follows

ExampleGeneric__1 [ X__t : TYPE ] :

THEORY
BEGIN

ZGenericSchema : [set[X__t]
  -> [# first : X__t,
      second : X__t,
      third : X__t #] -> bool] =
  LAMBDA (X : set[X__t]) :
    LAMBDA (s : [# first : X__t,second : X__t,third : X__t #]) :
      ...
      ...

Then we define the constants using the epsilon function.

first : [set[X__t] -> X__t] =
  LAMBDA (X : set[X__t]) : first(epsilon(ZGenericSchema))
  ...
  ...

Notice that the constant first is parameterised by a set. This is instantiated by any explicit instantiation of the generic parameters of first when it is used in term. In practise a generic constant will often be used without an explicit instantiation of its type parameters. We could just explicitly provide the instantiation in PVS or HOL whenever the constant appears in a term. But to maintain our goal of (where possible) providing a simple mapping from translated specification back to source, we choose instead to follow the constant definition with a further definition for each constant with a default value provided for the generic instantiation.

first__1 : X__t = first(fullset[X__t])
  ...
  ...

Notice that we have had to mangle the name slightly to differentiate the instantiated and uninstantiated form of the constant. The HOL representation is
not dissimilar. This representation of generic constants is by necessity somewhat
different and perhaps not particularly intuitive on first reading.

We come finally to the representation of the low level terms. In many ways
this is the most important aspect of the representation as during proof one is
working exclusively with these terms rather than the theory containers and it is
during interactive proof that the users intuition will be most useful.

The first and perhaps most serious problem in the term representation is the
disparity between what is considered a valid constant identifier in HOL, PVS and
Z. It is disappointing that even the API for HOL places restrictions on what is
a valid identifier. The API should allow the user to work at the level of abstract
syntax where it should be irrelevant what is the precise composition of an identi-
 fier. We are therefore forced in our embedding to mangle certain Z identifiers to
conform to the restrictions of the host notation. Z allows a particularly rich set
of identifiers particularly as regards names for infix operators. Name mangling
can therefore have a quite detrimental impact on program readability.

Z also supports a very flexible term syntax, allowing the user to introduce
new operators with one of a variety of fixities. In both PVS and HOL we need
to represent these richer syntactic constructs by simple prefix applications (al-
though infix constants, at least, can be defined in HOL). This means the term
will often be rather different in form but we have not found that this significantly
compromises readability.

4 Reusing Libraries: Supporting the Z Toolkit

So far we have considered the problem of embedding Z in the notation of a prover
such as PVS. This allows us to reuse the expertise of the tool constructors. We
would like also to re-use the work contained in the various libraries constructed
for the tools.

A particular area where this is pertinent to Z is the Z Toolkit. Z has an
extensive toolkit of definitions for mathematical constructs such as relations,
sequences and bags. We would like, as far as possible, to support the Z Toolkit
using existing libraries of definitions, theorems and tactics.

We have a choice, we can directly define the Z Toolkit in terms of elements
of available libraries or we can use the elementary Z Toolkit schema definitions
and then establish any relationships between the Z definitions and any available
libraries.

An advantage of the former approach is that we are able to make more direct
use of an available library of tactics and theorems. A disadvantage is the more
complicated translation. We must provide an explicit translation of all elements
of the Toolkit.

An advantage of the latter approach is that our translation need only support
the very core of the Z notation. We define the Z Toolkit by translating the Toolkit
schema definitions. A disadvantage is that we need explicitly demonstrate any
relationship with the definitions of an existing library. Proof can also be made
more difficult by the necessity for a certain amount of translation before a library
theorem or tactic may be used in a proof.

We have favoured the latter approach in our automated translation primarily
because of simplicity of translation and also because the existing HOL and PVS
libraries do not provide the exact functionality we require. The libraries all treat
relations, functions, bags and sequences as very different types, in Z they are all
just specialisations of the type of the relations.

5 Conclusions

In this paper, we have considered some of the problems of embedding one formal
notation into another. We have made particular reference to the embedding of
the Z notation into the notations of the HOL and PVS theorem provers, drawing
on experience gained implementing an automatic translation using the JavaLIL
tool.

We have tried to highlight what we believe are the particular problems in
supporting a translation that is both automatic and as far as possible complete.
Although state of the art, we have found HOL and PVS, in many ways, to be ill
suited to supporting such an embedding. In particular because of their lack of
support for user extension and configuration. In a companion paper, we consider
the problem of building an open and extensible theorem prover.

References

Press, 1996.
6. C. H. Pratten. An introduction to proving AMN specifications with PVS and
the AMN-PROOF tool. In 7th International Conference on Information System
In J. von Wright, J. Grundy, and J. Harrison, editors, Theorem Proving in Higher Order Logics (TPHOLs'96).
8. D. W. J. Stringer-Calvert, S. Stepney, and I. Wand. Using PVS to prove Z refine-
ment: A case study. In FME'97. Lecture Notes in Computer Science, Vol. 1313,
In D. Kapur, editor, Proceedings of the 11th International Conference on Automated
Deduction (CADE-11) Lecture Notes in Computer Science, Vol. 607, pages 748–
Building HOL90 Everywhere Easily
(Well Almost)

Elsa L. Gunter

Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA
elsa@research.bell-labs.com

Abstract. We describe how to change HOL90 so that it can build under all operating systems for which a full SML compiler exists. We also discuss some efficiencies, particularly separate compilation, that call for special compiler support and describe how they can be handled in the particular compiler, SML-NJ.

1 Introduction

Until recently, HOL90 has been mainly restricted to running on computers running some variant of Unix. With the advent of Linux, this is perhaps not too severe a restriction, since anyone with a PC and enough disk space can load Linux and then run HOL90 under that. However, the percentage of Unix users is still dwarfed by the users of other operating systems, and requiring someone to load and learn a new operating system just to use a tool like a theorem prover will prove too great a hurdle for many, and a considerable inconvenience for more. There have been some attempts to build HOL88 and HOL90 on MacOS with modest success, but at least for HOL90, they have not been robust. To the best of my knowledge, at the time of writing this paper, Isabelle-HOL is the only variant of HOL that builds on a Win32 operating system. In this paper, I describe the modifications to HOL90 to make it build under either a Unix operating system or a Win32 operating system (and presumably MacOS and any other operating system that SML-NJ runs under).

There are two main ways in which HOL90 has been altered to remove its dependence on Unix. The first is to eliminate all dependencies on Unix paths, and the second is to eliminate all its dependencies on shell scripts and makefiles. Unix paths may all be eliminated in favor of generic path constructions by using the various functions from the revised Standard Basis [5] structure OS.Path. We have included this structure as a substructure of Portable named OS.path. The main functions we use are
structure OS_path:
  sig
    ...
  val splitDirFile : string -> {dir:string, file:string}
  val joinDirFile : {dir:string, file:string} -> string
  val splitBaseExt : string -> {base:string, ext:string option}
  val joinBaseExt : {base:string, ext:string option} -> string
  val concat : string list -> string
    ...
  end

It is unfortunate that with the new Standard Basis, paths were not made a separate type, with other operating system functions, including those for IO, taking paths instead of strings as arguments. This would have helped the user to be sure they were writing code that was operating system independent.

Eliminating HOL90’s dependence on shell scripts or makefiles is somewhat more complicated, and we shall discuss it in more detail in the rest of this paper. The basic strategy is to control the entire build from a single SML file. Eliminating HOL90's dependence on shell scripts and makefiles might be unnecessary for building HOL90 on Win32 operating systems, since there exists a gnu-make for these and it is possible to acquire shell languages such as the Korn shell for these. However, for MacOS, this dependence is still a problem. Moreover, if HOL90 depends only on SML and its new Standard Basis, then porting HOL90 to a new operating system should only require using a port of SML to that operating system. Also, there is a distinct advantage to placing as few system requirements on potential users as possible. The more pieces the user must acquire (possibly through purchase), install, and learn to use, the higher the hurdle they face to using the end system.

A related issue is our dependence on a particular implementation of SML. There are two barriers to HOL90’s being independent of the particular implementation of SML. One barrier is that different compilers support different mechanisms for loading, separate compilation, and exporting an executable. The other barrier is that HOL90's anti-quotation mechanism requires special support from the compiler. SML-NJ has such a special modification built in. However, it is not a part of the Standard [4], and any compiler is free to not provide such support, or to provide a different mechanism.

To handle the first barrier, we provide different loading files for different versions of SML. Right now, these consist of a generic one that relies only on use, one that is specialized to SML-NJ to the extent that it can switch between compiling and interpreting code, and uses its ability to export executables, and one that uses CM, the compilation manager of SML-NJ. Throughout, I have tried to make changes that were either built from the new Standard Basis, and therefore available in all compliant compilers, or that tightly encapsulated those aspects that were compiler dependent in such a way as it should be readily possible to substitute other functions from other compilers to perform the same core functionality.
The second barrier, that of anti-quotation, is a more difficult one. If the top-level loop of SML cannot give special treatment to the quotation, then a procedure is needed to translate the SML syntax augmented by quotations and anti-quotations into pure SML syntax with the the quotations and anti-quotations being translated into an internal type in SML (namely \texttt{frag lists}). This procedure needs to pass the converted strings to the front end of SML. Without special help from the compiler, this forces the the procedure to be executed as a separate process in the operating system and to pass the translated string to the Standard In of the SML process running the core of SML. This calls for operating system support for inter-process communication. At present I do not know how to handle inter-process communication in an operating system independent manner without using special support from the SML compiler. Somewhat awkwardly, the only SML compiler we know that provides sufficient support for inter-process communication on all operating systems is SML-NJ, and this compiler also provides sufficient support for quotation and anti-quotation to make a preprocessor unnecessary. At present, the approach we are taking is that the user may specify whether HOL90 is to be built with a preprocessor or not. If they choose to use a preprocessor, then they need to be running a Unix operating system, or using the SML-NJ compiler. If they choose not to use a preprocessor, then they can use other operating systems, but they must also use a compiler (such as SML-NJ) that provides special support for quotations and anti-quotations.

2 A make-less Build

Building HOL90 has several phases which cannot readily be merged into one single SML session. To build the core system there are layers of compiling data structures and functions, and there are layers of building theories. In addition to the core, there are libraries containing their own layers of theories and code. For the theories to be built, the code that is used in the proofs of the theorems must exist, but, in turn, for much of the code for proving theorems to exist, certain definitions and theorems in certain theories must exist. This alternation of compiling code and building theories does not in itself prevent the build from occurring within a single SML session. Unfortunately, the way theory dependencies are created in HOL90 does prevent all the theories from being built in a single session. In HOL90, after the first theory \texttt{min} is created, HOL90 is always in a theory (the "current theory"), and when a new theory is created it becomes the current theory, and the previously current theory becomes one of its parents. However, the dependency graph of the theories of HOL90 is a dag and not a linear order. If we were to linearize the theory dependency dag, we would introduce false dependencies. This would be a bit confusing, but mathematically harmless for those theories in the core. However, it is not even possible with the set of all theories from all the libraries; there exist theories from different libraries which are incompatible. If we were to build all of HOL90, libraries included, from within a single SML session, this would introduce a linear ordering on all the theories associated with HOL90 (for all those theories it could build before
it encountered a theory that was inconsistent with one previously loaded). To avoid introducing unnecessary or undesirable theory dependencies and, indeed, to allow all the libraries to be built, we need to be able to support multiple SML sessions.

The way this layering was handled in the earliest versions of HOL90 was by having a shell script that called SML, loaded enough code to start building theories, exported an image, used the exported image repeatedly to build each theory in the first layer of theories, then called the exported image to load code that depended on those first theories, exported a new image, used that new image to build the next layer of theories and code, and so on until all the layers were built. This methodology was extended to the libraries, with each library having a (possibly empty) theory layer, followed by a (possibly empty) code layer.

In order to allow HOL90 to run on multiple implementations of SML, starting with version 9, a front end was added to translate quotations and anti-quotations into standard SML (with frag lists). This added an additional layer on to the make process. The front end is a separate process that communicates with HOL90 via a Unix pipe. Thus the building of HOL90 creates two separate executables and a collection of theory files (and library description files for a full build). If one of these becomes out of date, it does not necessarily mean that the rest are out of date. In the earliest versions of HOL90, it was only possible to specify at the beginning of the build whether to rebuild all the theory files or to reuse them all. This can lead to a good bit of unnecessary re-execution rebuilding theories that are not out of date. When the move was made to include a front end, the decision was made to also move to the use of make and makefiles to control the build process. This allowed for better and more automatic control over which components were rebuilt on any given build.

Using makefiles, make [2] determines whether a given file is out of date by comparing the system modification time of the file in question with the system modification times of the list of files upon which the given file is specified to depend. If it finds the given file to be older than any of the files upon which it depends, then the file is considered to be out of date and the given code for rebuilding the file is executed.

This notion of being out of date is not accurate for theories in HOL. Each HOL theory depends upon a set of parent theories as well as the SML code used to create it. If the code files have been modified more recently than the theory files, then the theory is out of date. However, just because a parent theory has been modified does not necessarily mean that the theory in question is out of date. It depends on the nature of the modification. The typical way in which a theory is modified is by having new definitions and theorems added to it. This kind of modification does not render any of its dependents out of date. This is because it has changed nothing that already-existing dependent theories actually depend upon. The dependence of one theory upon another arises by the dependent theory using a constant defined in an ancestor theory. If a theory uses a constant defined in an ancestor theory, then the validity of the theory depends upon the definition of the constant. The only way the definition can
be changed is if the ancestor theory is recreated. For HOL to know of a theory whether the definitions of its ancestors have changed would require each theory to record all the definitions that it uses from its ancestors. This would lead to space-consuming duplication. As a result, HOL 90 uses a conservative approximation based on the creation times of theories. Each theory records the time at which it was first created, and the creation time of each parent theory. As a theory has new definitions and theorems added to it, its creation time is not altered. Only deleting the theory files and creating the theory anew alters its creation time. This is also the only way that a definition within a theory may be changed. Therefore, it is safe to assume that if the creation date of a parent has not changed, then the given theory is not out of date with respect to that parent, and if it has changed then it is.¹

To correctly determine whether theory files are out of date requires parsing the files to find the creation dates of the files and their parent theories. This is out of the scope of make. The easiest place for the determination of whether a theory file is out of date, at least with respect to its ancestor theories, is within HOL 90 itself where all the infrastructure already exists. Testing whether a theory is out of date with respect to parent theories is a calculation already performed by HOL 90 when a theory is loaded. Calculating whether it is out of date with respect to its source files may be done using the SML Standard Basis functions

\[
\text{OS.FilenSys.modtime : string} \to \text{Time.time}
\]
\[
\text{Time.<} : \text{Time.time} \times \text{Time.time} \to \text{bool}
\]

We have encapsulated the composition of these two functions in the HOL 90-defined function

\[
\text{Portable.OS_ops.more_recent_than :}
\]
\[
\{\text{make_files : string list, results : string list}\} \to \text{bool}
\]

So far we have explained that HOL 90 is the easiest place to accurately test whether a given theory is out of date, but we have also explained that the theory files cannot, as HOL 90 currently works, all be built in a single process. One solution would be to alter HOL so that when a new theory was created, the user would have to specify whether the previous theory was to be a parent or not. This path is possible and potentially the right way to go in the long run but fraught with complications that require a major rewrite of the core of HOL. The other solution, which we shall pursue here, is to accept that there will need to be multiple processes and to determine how to manage them from within SML.

HOL 90 may be built in one SML process if all the theory files for both the core and all the libraries exist and are up to date. But they don't actually all have to exist and be up to date before the SML process begins; for the core

¹ If an ancestor theory adds a new constant of the same name as one previously defined in a dependent theory, then the dependent theory will be rendered invalid and cannot subsequently be loaded. However, we do not consider this as rendering the dependent theory out of date because rebuilding the theory will not solve the problem. The rebuild will fail when the duplicate constant definition is encountered.
theories, they just need to exist and be up to date by the time they need to be installed in the system theory graph and used by code, and for library theories, they only need to exist and be up to date by the end of the build. Therefore, the approach we take to building HOL90 is to have a top-level process that controls the build and creates the core HOL90 executable. After it has finished building a layer of code and needs a layer of theories, it tests each of those theories to see if it is out of date, and for those that are out of date it starts a sub-process to rebuild them. Once all theories for the next layer are known to be up to date, the top-level process proceeds to build the next layer of code.

To handle the layered build of HOL90, we need to be able to create executables that can rebuild the theories that are out of date. To support this, we have added a subdirectory tmp to the HOL90 bin directory. This directory will be used for holding temporary executables and related files needed for building theory files and libraries. Each time we reach a layer where we need new theories, first a determination is made if all the needed theories are up to date. If they are, then the build moves on to loading them and using them as needed. If there are theory files that out of date, then we export into the tmp directory a copy of the SML process to be used for rebuilding the out-of-date theories. The executable that we export needs to be used for rebuilding the out-of-date theory files, while the process from which it was exported must not rebuild any theories. At least for SML-NJ, this poses a minor problem. When an executable is exported and subsequently executed, it starts by executing whatever code remained to be executed in the process that did the export. In the process that does the exporting, the remainder of the computation is to call an external executable to rebuild the needed theories and then to build the rest of the system. The process that is exported needs a way not to do that computation, but to rebuild the specified theory instead. The method we use for handling this problem in SML-NJ uses the fact that the function that does the export returns true in the process exported, and returns false in the process that does the exporting. (We would require something similar of other SML compilers.) Immediately following the export, the next computation to be done can depend upon whether the computation is being done in the exported process or the original process. In our case, we choose to have the exported process use a specific file load_file.sml to be found in the directory tmp and then exit. The top-level process skips this computation and proceeds on with the rest of the build. The file load_file.sml typically doesn’t exist when the secondary process is exported. After the export is done, for each theory file that needs rebuilding, we write into load_file.sml instructions for building the theory and then call the exported executable. The secondary process then executes the instructions for building the theory and exits. Upon the termination of the secondary process, control is returned to the top-level process. At the end of rebuilding all the theories for a given layer, we clean up, deleting the executable and the load file.

There are two ways to avoid the use of a load file. One way is to make the exported process directly execute the code for rebuilding the theory. This requires creating a separate executable for each theory that needs rebuilding. In
our method, we create only one executable per layer. The other method is to pass
the instructions to the exported executable using the command-line arguments.
There is no support in the Standard Basis for determining what the command-
line arguments are, so this solution would be compiler specific. Also, the use
of a load file seems to us more flexible for general use. The infrastructure de-
scribed above was created to accommodate the build process, but it is available
to the end user who may find other uses of it. As a result we tried to find the
most flexible solution. This basically finishes the description of the infrastruc-
ture necessary for building the core HOL90 system, including its theories. This
infrastructure is also sufficient for building all of the theories in the libraries,
assuming that we continue to load libraries using the old loading mechanisms,
and in particular if we are not concerned about precompiling the code in the
libraries.

3 Precompilation

To date, there is no uniform method for providing precompilation across all SML
compilers. By precompilation, we mean the ability to store compiled code in files
on disk so that it may be loaded and executed without having to recompile the
source code. There is no support built into the Standard Basis for precompilation.
Therefore, the best we can do is to create a solution that is compiler specific but
operating system independent and try to encapsulate the compiler-specific parts
while making the rest of the mechanism as general as possible. The compiler we
have fixed upon is SML-NJ with precompilation provided by the Compilation
Manager (CM) [1].

CM was designed to support smart recompilation (like make, but specialized
to SML). It does so by caching units of compilation and information about their
dependencies. Its ability to support precompilation is a side-effect of this. The
main function from CM we use is

\[
\text{CM.make'}: (\text{string} \ast \text{bool}) \rightarrow \text{unit}
\]

The first argument to this function is the name of a file (which we shall refer to as
the CM file) that contains a description of the source files to be compiled (and an
optional description of what code is to be exported to the top-level environment).
The second argument tells whether all modules should be re-executed each time
they are re-linked. In our case we always want this second argument to be false.

CM needs to know the complete set of source code dependencies. These are
described in the CM file. For source code that has already been is given in
previous CM files, the name of the CM files should be listed in the current CM
file. CM keeps a search path for finding CM files given within other CM files.
If the CM file name given has no path extension, then CM will look for the file
first in the directory of the current CM file, and then in the directories given in
the search path. If there is a path extension on the CM file name, if the path
is relative, then it is interpreted relative to the directory of the current CM file,
and if it is an absolute path then it just uses the file found in the directory given
by that path. For the new source code, the SML files are listed. There is no searching for these files, but if an absolute path is not given, the path (including the empty path) is interpreted relative to the directory of the current CM file, just as with CM files.

Our strategy for supporting precompilation is as follows. Our unit of precompilation is an HOL90 library and all code supplied with the system is contained in some library (including all of the core system). Each library corresponds to a CM file. When a library is created (via `new_library`) a library description file and a CM file are written into the directory at the head of the library description search path (given by `Global.library_path`). Theory files get remade if necessary and source code gets precompiled if necessary in an external copy of the HOL process when the library is created, and in the current process each time the library is loaded.\footnote{There is one exception to this. The library containing the collection of all system library descriptions (including its own) cannot be precompiled when it is created, for this leads to an infinite regress. Its gets precompiled more explicitly during the build of the system. The core library that defines all the library handling functions, among other things, is sort of an exception as well in that its source code always is (and must be) precompiled before the library that contains it can be created.} The library search path is included in the CM search path, and every time the library search path is updated, the CM search path is updated in a similar manner.

When a user creates a library, they must specify a collection of files and paths. To make these specifications portable across multiple operating systems, the paths are given as lists of strings of path arcs and files are given using the type

\begin{verbatim}
{dir : string list, base : string, ext : string option}
\end{verbatim}

For example, the Unix file ".../prim_hol/src/tactic.sml" would be given by

\begin{verbatim}
{dir= ["..","prim_hol","src"], base= "tactic", ext= SOME "sml"}.
\end{verbatim}

Paths specified using the above type cannot be specified as absolute; they will always be interpreted as relative. The value given for `path` will be treated as relative to some one of the paths in the library search path. The resulting absolute path we shall call `<library_path>`. The files for remaking theories will be treated as relative to `<library_path>/theory/src` (using Unix syntax). The code files will be treated as relative to `<library_path>/src`, and the help paths will be treated as relative to `<library_path>/help`. From this information, together with the list of parent libraries, HOL90 can automatically generate a CM file for the given library.

Precompilation requires disk space. It may be possible to spare the disk space for precompiling the code for HOL90 libraries without end users having the personal disk space to store precompiled binaries from personal libraries. Therefore, we supply a flag

\begin{verbatim}
SysParams.precompile : bool ref
\end{verbatim}
that allows the user to toggle whether precompiled binaries will be generated and stored when the code for a library is created or loaded. If the user needs to load a personal library and not have binaries stored in their personal space, then they may set `SysParams.precompile` to `false`. If they later wish to have personal libraries precompiled, they will need to first set `SysParams.precompile` back to `true`.

The ability to toggle between precompiling and not precompiling leads to several anomalous circumstances. The first is when HOL is set to not precompile and the user wishes to load a library which has been precompiled. In this instance, the user still wants to load the precompiled code. Therefore, each library carries with it a flag telling whether it has been precompiled. If the flag is `true`, then the precompiled code will be loaded, even if the HOL flag for precompiling has been set to `false`. The next difficulty arises when the HOL precompile flag has been toggled from `false` to `true`. In this case, we assume that if the user loads a library, they wish it to be precompiled, even if it was not when it is first created. This may not be possible, however. For example, the library may be in another user's directory. Therefore, if precompilation fails for any reason we gracefully back off to ordinary loading of the library with no precompilation. If a library cannot be precompiled, then no library that depends on it can be precompiled either. Therefore, we assume that if a library has been loaded without being precompiled, then it should not be precompiled (at least at this time), and no library that depends on it will be precompiled either.

There are three instances in the core code of HOL90 that are specialized to using CM for precompilation. Two are visible to the end-user and the other is not externally visible. We have already seen that `Globals.library_path` is specialized to using CM in that updating it also updates the CM search path. The externally visible function `SysParams.load_precompile` takes both the CM file and the list of files that need to be loaded. It attempts to load the needed files using CM. This will attempt to reuse any existing precompiled units, and precompile any that don't exist. If CM fails, then `SysParams.load_precompile` backs off to using the standard methods of loading files. It is for this that the list of files is given. The other function is called by the function `Library.newLibrary`. The only effect this function has is to write out a CM file summarizing the code dependencies of the library. Excluding enhanced efficiency and the presence of additional files on disk, none of these instances of code specialized to CM have any effect on the computations of HOL90 nor do they have any real dependence on CM's existence.

4 Where Things Stand

We have made HOL90 build through the end of the core system and successfully precompiled several of the libraries. The remaining libraries still require processing to replace Unix specific paths by operating system independent paths. A few libraries (such as the reals) export an augmented HOL90 for building the theories in the library. These libraries will need to be modified to fit into the new
built process. Since our build process also does an export, it should be possible to merge the two, but these libraries will pose a challenge for the build process.

Barring complications that may arise from handling such libraries, the build process is now quite simple. To install, one needs only load a single sml file. This file in turn loads either a CM file or an SML file to build the core system up through the library mechanisms, then installs the theory min, then loads a file to build the system-library library, then loads the library HOL restoring the usual HOL90 initial state, and finally exports a copy and exists. If the person building the system would prefer a different initial state, they must only choose to load a different library (or set of libraries) before the final export.

We have not yet built support for a front-end preprocessor with the combination of SML/NJ and Win32. The preprocessor that comes with version 10 of HOL90 is written in C using Lex and Yacc. To avoid requiring the user of HOL90 to have a C compiler, we plan to rewrite the preprocessor in SML using ML-Lex and ML-Yacc. Then the interprocess communication would also be handled by SML.

Rigorous testing of the whole build process for HOL90 will be required.

References

Program Composition in Coq-Unity

François Marques
CNET, France Telecom, Lannion, France
francois.marques@cnet.francetelecom.fr

Abstract. Unity has been implemented in many theorem provers. In this paper, we consider Coq-Unity, the embedding of Unity in Coq. A primary characteristic of this system is that it has been proved correct and complete, and it keeps compositability of temporal properties. However, no support is offered in order to change the set of variables involved in the definition of a state. This makes the development of reusable libraries difficult.

After a brief presentation of Unity and its embedding in Coq, we will suggest a solution to get around these restrictions. Finally, we apply this method to a classical example: the Alternating Bit Protocol.

1 Introduction

Interactive proofs of properties of a complex system are difficult to obtain. There are two ways to tackle these proofs. One is to split complex systems into smaller ones. Another is to decompose properties into easier ones. Nevertheless, interactive theorem proving is viable only if proofs (and systems) are reusable.

The Unity framework provides rules to split both systems and properties. Reusability is more problematic: it requires the possibility to extend the set of variables used by a component to the state used by the full system. However, the validity of some properties depends on the set of variables defining the program state. For example, assume property \( p \) thus defined: for all variables, their value will become positive.\(^1\) The program with unique transition \( \{ x := x + 1 \} \) verifies the property \( p \) if the set of variables is \( \{ x \} \), but does not if it is \( \{ x, y \} \). Nevertheless, most properties are preserved when the state is extended.

Coq-Unity [5] is an embedding of Unity [3] in Coq developed by Heyd and Crégut. This system provides modularity and composition of proofs of properties. Contrary to HOL-Unity [1], this framework enables an expressive variable description using dependents types [8]. The authors of this embedding have depicted a way to provide only a partial description of the state of a program, which could then be extended to fit the specific needs of the use of the component [4]. This method can be applied recursively, but is heavy because it requires the use of many abstract coercion functions. These ones degenerate to the identity function when they are specialized.

\(^1\) \( p \) is expressible in Coq-Unity: \( p \vdash [s:\text{state}] (\forall v:\text{variable}. (s \ v) \geq 0) \).
This article presents another solution. The component is described via a state reduced to the useful variables and then embedded in another state defining more variables with the help of a function $\varphi$ satisfying some conditions. The component properties are proved on the reduced state. We give theorems that ensure these properties are lifted to the extended state.

After a brief presentation of Coq-UNITY, we expose our solution to support program state extension and apply it to a classical example: the Alternating Bit Protocol. The proof shown here is elegant because it is based on the symmetry of the protocol.

2 Coq-UNITY

UNITY consists in two components: a program notation and a logic whose purpose is to prove some temporal properties of the programs.

In fact, the embedding of UNITY in Coq is an extension of the original theory. Modifications were introduced to obtain a complete theory that preserves properties by program composition.

Coq-UNITY is implemented in Coq [6]. Coq is an interactive theorem prover based on the Calculus of Inductive Constructions. It is a higher order typed lambda-calculus with primitive mechanisms to define inductive types with their constructors, destructors and elimination theorems.

In the following, we will use a pseudo-mathematical notation, which is more compact and easier to read than Coq syntax. We give below an overview of the Coq-UNITY system.

2.1 Basic Definitions

variable set: this is a set of typed variables. Generally, we write it $V$;

state: a state is a mapping from variables to values of the corresponding type.

If $V$ is the variables set, we denote the set of states by $S_V$;

transition: a transition is a function from states to states.

2.2 Program Description

A Unity program description is a four-tuple $P = (V, I, T, C)$

- a set $V$ of (typed) variables,
- a set $I \subseteq S_V$ describing the initial states of the program,
- a set $T$ of program transitions,
- a set of transitions $C$ describing the context of the program.

The context is the set of transitions that can be used by other programs executed in parallel. So, two programs $P_1 = (V, I_1, T_1, C_1)$ and $P_2 = (V, I_2, T_2, C_2)$ can be composed if $T_1 \subseteq (T_2 \cup C_2)$ and $T_2 \subseteq (T_1 \cup C_1)$.

An execution of the program $P = (V, I, T, C)$ is defined by:
• an initial state \( s \in I \)
• an infinite sequence of transitions \( t_i \in T \cup C \) such that for each transition \( t \in T \) we have: \( \forall j \exists i (i > j) \land (t = t_i) \). This constraint is called the fairness condition.

This definition leads us to define reachable states of a program. The set of reachable states of the program \( P = (V, I, T, C) \) is the closure of \( I \) by the transitions of \( T \cup C \) (i.e. a state \( s \) is reachable by \( P \) if there exists an execution of \( P \) that leads to \( s \) ). We write \( \mathcal{R}_P \) the set of reachable states of the program \( P \):
\[
s \in \mathcal{R}_P \equiv (s \in I) \lor \exists t \exists s'. (t \in T \cup C \land s' \in \mathcal{R}_P \land s = t(s'))
\]

2.3 Description of the Unity Logic

This is a subset of a linear temporal logic. It defines safety and progress operators. Assume \( P = (V, I, T, C) \) is a program and \( u \) and \( v \) are state predicates. There are three basic operators.

• \texttt{unless}_P \) is a safety operator. \( u \texttt{unless}_P v \) means that the property \( u \) will remain \texttt{true} unless \( v \) holds.
  \[
u \texttt{unless}_P v \equiv \forall t \forall s . (t \in T \land s \in \mathcal{R}_P ) \Rightarrow (u(s) \land \neg v(s) \Rightarrow u(t(s)) \lor v(t(s)))
\]

\( \texttt{stable}_P \) and \( \texttt{invariant}_P \) are safety operators derived from \( \texttt{unless}_P \):
\[
\texttt{stable}_P u \equiv u \texttt{unless}_P \texttt{false} \\
\texttt{invariant}_P u \equiv (\texttt{stable}_P u) \land \forall s (s \in I \Rightarrow u(s))
\]

• \texttt{ensures}_P \) is the elementary progress operator. \( u \texttt{ensures}_P v \) is similar to \( u \texttt{unless}_P v \) but it also requires the existence of a transition transforming any state satisfying \( u \) into a state satisfying \( v \).
  \[
u \texttt{ensures}_P v \equiv (u \texttt{unless}_P v) \land \exists t . (t \in T \land \forall s (s \in \mathcal{R}_P \Rightarrow (u(s) \land \neg v(s) \Rightarrow v(t(s))))))
\]

• \texttt{leadsto}_P \) is the other progress operator. It is defined as the transitive and disjunctive closure of \( \texttt{ensures}_P \).

The Unity logic provides some derived rules. For example the \( \texttt{unless-cancel} \) rule is:
\[
\frac{(u \texttt{unless}_P v) \quad (v \texttt{unless}_P w)}{(u \lor v) \texttt{unless}_P w}
\]

3 Embedding of Program States

3.1 Introduction

Suppose we have a program \( P = (V, I, T, C) \) where \( V = \{x, y\} \), and one wants to extend \( V \) by introducing a new variable \( z \). Cregut and Heyd have presented in [4] a method to do so.
In their method, V and its visible members are declared as Coq variables. Then, for each variable, we introduce two functions and the hypothesis that they are reciprocal. For this basic example it corresponds to:

Variable V : Set, domain : V → Set.
Variable x : V.
Variable ext_x : (domain x) → nat, inj_x : nat → (domain x).
Hypothesis H1_x : ∀ n : nat. (ext_x (inj_x n)) = n.
Hypothesis H2_x : ∀ n : (domain x). (inj_x (ext_x n)) = n.

Furthermore we need a function to differentiate the variables, and another hypothesis for each variable:

Variable case : ∀ v : V. (domain x) → (domain y) → (domain v).
Hypothesis case_x : ∀ x : x → y. (case x fx fy) → fx.

In the program specification, we refer to variable values through their corresponding functions, and we use the hypothesis in the proofs. Then, when we need to extend the variable set, we must define the new set of variables, and specialize the program in it.

This method has some limitations. It is tedious and requires to declare the program in accordance with it; if we have declared the program in another way, we can not apply the result of this method.

3.2 A New Way

Assume we have the original program P = (V, I, T, C) and the extended one P' = (V', I', T', C'). S and S' are respectively the set of states defined over V and V'. We introduce a function φ from S' to S corresponding to the omission of the added variables.

For the previous example, φ will be the function:

φ : S' → S, s' → s : s(x) = s'(x) ∧ s(y) = s'(y)

We consider now the four following conditions:

- **(C1)** φ maps each P' initial state to a P initial one:
  \[ ∀ s'. (s' ∈ I' → φ(s') ∈ I) \]

- **(C2)** for each P'-transition t', there exists a transition t of the program P such that φ ◦ t' = t ◦ φ:
  \[ ∀ t'. (t' ∈ T' → ∃ t. (t ∈ T ∧ ∀ s'. (s' ∈ S' → φ(t'(s')) = t(φ(s'))))) \]

- **(C3)** for each transition t' of the P'-context, there exists a transition t of the context of the program P such that φ ◦ t' = t ◦ φ:
  \[ ∀ t'. (t' ∈ C' → ∃ t. (t ∈ C ∧ ∀ s'. (s' ∈ S' → φ(t'(s')) = t(φ(s'))))) \]

- **(C4)** for each P-transition t, there exists a transition t' of the program P' such that φ ◦ t' = t ◦ φ:
  \[ ∀ t. (t ∈ T → ∃ t'. (t' ∈ T' ∧ ∀ s'. (s' ∈ S' → φ(t'(s')) = t(φ(s'))))) \]
In the case of program state extension, these properties are very easy to check and can be proved mechanically. We can now state the following theorems:

**Theorem 1** if \( \varphi \) satisfies (C1), (C2) and (C3) then it transforms \( \mathbb{P}' \)-reachable states into \( \mathbb{P} \)-reachable ones:

\[
\forall s'. (s' \in \mathcal{R}_{\mathbb{P}'}) \Rightarrow \varphi(s') \in \mathcal{R}_{\mathbb{P}}
\]

**Theorem 2** if \( \varphi \) satisfies (C1), (C2) and (C3) then for any S-predicates \( u \) and \( v \), we have:

\[
\text{u unless}_{\mathbb{P}} v \Rightarrow u \circ \varphi \text{ unless}_{\mathbb{P}'} v \circ \varphi
\]

\[
\text{u ensures}_{\mathbb{P}} v \Rightarrow u \circ \varphi \text{ ensures}_{\mathbb{P}'} v \circ \varphi
\]

**Theorem 3** if \( \varphi \) satisfies (C1), (C2), (C3) and (C4) then for any S-predicates \( u \) and \( v \), we have:

\[
\text{u leadsto}_{\mathbb{P}} v \Rightarrow u \circ \varphi \text{ leadsto}_{\mathbb{P}'} v \circ \varphi
\]

4 Application: The Alternating Bit Protocol

The Alternating Bit Protocol is a communication protocol adapted for data transmission through an unreliable channel. It was first described in [7].

This protocol has been formally verified many times in numerous systems including the Calculus of Constructions [2]. However, we give here an elegant modular proof using state extension and based on the symmetry of the protocol.

4.1 Description

The Alternating Bit Protocol specifies the behaviour of two sender-receiver SR1 and SR2, a channel Channel12 from SR1 to SR2 and a channel Channel21 from SR2 to SR1. The intuitive idea is that the sender dispatches the same message until it receives the corresponding acknowledgement (an acknowledgement becomes a new message receipt). A message is thus repeated many times in the channel. In order to differentiate two identical messages from a repetition of an unique message, the sender adds a bit to each message. This bit value changes each time the sender treats a new message.

We call *pack* the data composed with a *bit* and a *message*, and we write a *pack* as a couple (bit, message). We define the following functions in order to manipulate *pack* data:

- \( \text{pack}(b, m) \) returns the *pack* data composed with the *bit* \( b \) and the *message* \( m \);
- \( \text{bit}(p) \) returns the value of the *bit* contained in the *pack* \( p \);
- \( \text{message}(p) \) returns the value of the *message* contained in the *pack* \( p \).
4.2 Coding the ABP

Unreliable Channels. We model a channel with a FIFO structure. Three functions are associated to this structure:

- **write**\( (p, C) \) returns the channel obtained by adding the *pack* \( p \) to the channel \( C \);
- **read**\( (C) \) gives the oldest *pack* written into the channel \( C \);
- **cut**\( (C) \) returns the channel obtained by taking off the older *pack* from the channel \( C \).

To model the unreliability of the channel, we add in the context of the program transitions that simulate loss of messages. However, we must limit losses otherwise the channel may lose all the messages. Therefore, we associate a variable CanLose to the channel, to specify that this channel can or cannot lose messages anymore. We add a transition tStopLose in the program to put this variable value to *false* and the transitions in the context are constrained by this variable value. The context of the program can read the channel only when CanLose value is *true*.

The fairness condition ensures that tStopLose will be executed, so the channel can not indefinitely lose messages. Naturally, when we read a *pack* in the channel we put back this variable value to *true*, allowing the channel to lose messages again.

Message Queues. A queue is, like a channel, a FIFO structure. Four functions are associated:

- **put**\( (m, Q) \) returns the queue obtained by adding the *message* \( m \) to the queue \( Q \);
- **head**\( (Q) \) gives the oldest *message* put into the queue \( Q \);
- **pop**\( (Q) \) returns the queue obtained by taking off the older *message* from the queue \( Q \);
- **last**\( (Q) \) gives the last *message* put into the queue \( Q \).

We want to prove that having a sequence in an input queue, at some point, the same sequence will be in the corresponding output queue. To do that, one can choose: either to memorize messages written in the output queue or to prevent reading in the output queue. We have chosen the second solution.

Variables. The sender-receiver \( SR_{i,(i \in \{1,2\})} \) has two queues: one for the messages to send and another for the received ones. We call these queues respectively SQueue\(_i\) and RQueue\(_i\).

We need also two boolean variables. One to memorize the *bit* it must send with a *message* in a *pack*, and another one to know whether the received *pack* corresponds to a new *message* or to a repeated one. We call these variables AltBit\(_i\) and CheckBit\(_i\).

Furthermore, there are two unreliable channels Channel\(_{12}\) and Channel\(_{21}\), and the associated variables CanLose\(_{12}\) and CanLose\(_{21}\).
Transitions. For each $i, j \in \{1, 2\}$ with $i \neq j$, we define the transitions:

$$t_{\text{Read}}_i \equiv \text{CanLose}_i \leftarrow \text{true}$$

$$\wedge \begin{cases} \text{if } \text{bit(read(\text{Channel}_j))} \neq \text{CheckBit}_i \text{ then} \\
R\text{Queue}_i \leftarrow \text{write(message(\text{Channel}_j), R\text{Queue}_i)} \\
\wedge \text{CheckBit}_i \leftarrow \neg \text{CheckBit}_i \\
\wedge \text{AltBit}_i \leftarrow \neg \text{AltBit}_i \\
\wedge \text{SQueue}_i \leftarrow \text{pop(SQueue}_i) \\
\end{cases}$$

$$t_{\text{Write}}_i \equiv \text{Channel}_j \leftarrow \text{write(pack(AltBit}_i, \text{head(SQueue}_i)), \text{Channel}_j)$$

$$t_{\text{StopLose}}_i \equiv \text{CanLose}_i \leftarrow \text{false}$$

Some Notations. For convenience, we introduce three definitions. Assuming $t$ is a transition, $\text{Var}$ is any variable, $\text{Queue}$ is a queue variable, $\text{Channel}$ is a channel variable and $\text{CanLose}$ is the corresponding variable, we write:

$$\text{DontChange}(t, \text{Var}) \equiv \forall s. (s \in S \Rightarrow t(s(\text{Var})) = s(\text{Var}))$$

$$\text{MayWrite}(t, \text{Queue}) \equiv \forall s. (s \in S \Rightarrow t(s(\text{Queue})) = s(\text{Queue}) \vee \exists m. (t(s(\text{Queue})) = \text{put}(m, s(\text{Queue}))))$$

$$\text{MayRead}(t, \text{Channel}) \equiv \forall s. (s \in S \Rightarrow t(s(\text{Channel})) = s(\text{Channel}) \vee (\text{transit}(s(\text{Channel})) = \text{cut}(\text{Channel}) \wedge \text{CanLose} = \text{true}))$$

4.3 Proof of the Protocol Correctness

The proof is based on the symmetry of the protocol. It is structured so that:

- We prove that a message in the $\text{SQueue}_1$ will be written in the $\text{RQueue}_2$. To do that we only need a half of a protocol.
- We embed the half-protocol into the complete protocol in two ways. So, we get the previous result lifted in the complete protocol into two symmetric properties.
- Using these two properties we prove that when $\text{SR}_1$ sends a packet, the message contained in the pack will be written in $\text{RQueue}_2$ and $\text{SR}_1$ will receive the corresponding acknowledgment, thus it can proceed with the next message.

Below, we give a sketch of the proof and of the coding used.

The Half Protocol. The half protocol is the program $P_1 = (V_1, I_1, T_1, C_1)$ corresponding to half of the complete program (see fig. 1):

$$V_1 \equiv \{ \text{AltBit}_1, \text{AltBit}_2, \text{CheckBit}_2, \text{Channel}_1, \text{CanLose}_{12}, \text{SQueue}_1, \text{SQueue}_2, \text{RQueue}_2 \}$$

$$I_1 \equiv \{ s \mid s(\text{Channel}_{12}) = \emptyset \}$$

$$T_1 \equiv \{ t_{\text{Write}}, t_{\text{Read}}, t_{\text{StopLose}} \}$$

$$C_1 \equiv \{ t \mid \forall x \in \{ \text{AltBit}_2, \text{CheckBit}_2, \text{CanLose}_{12}, \text{RQueue}_2 \} \Rightarrow \text{DontChange}(t, x) \}$$

$$\wedge \text{MayWrite}(t, \text{SQueue}_2) \wedge \text{MayRead}(t, \text{Channel}_{12}) \}$$
Grayed variables are omitted in program P1

Fig. 1. The half-protocol schema

Then we prove that whatever the SQueue1 head value may be, it is eventually written in RQueue2. More exactly:

Assuming b, in, out and x are values, for each $i, j \in \{1, 2\} \text{ with } i \neq j$, we call $u_i$ and $v_i$ the state predicates defined thus:

$$u_i(s) \equiv \text{head}(s(\text{SQueue}_1)) = x \quad v_i(s) \equiv \text{head}(s(\text{SQueue}_1)) = x$$

$$\land s(\text{AltBit}_i) \neq s(\text{CheckBit}_j) \quad \land s(\text{AltBit}_i) = s(\text{CheckBit}_j)$$

$$\land s(\text{AltBit}_i) = b \quad \land s(\text{AltBit}_j) \neq b$$

$$\land s(\text{SQueue}_j) = \text{in} \quad \land s(\text{SQueue}_j) = \text{pop(in)}$$

$$\land s(\text{RQueue}_j) = \text{out} \quad \land s(\text{RQueue}_j) = \text{put}(x, \text{out})$$

Then we have:

**Lemma 1** $u_1 \text{ leadstop}_P, v_1$.

To prove the lemma, we use the fact that the channel value is the concatenation of a sequence of $\text{pack}(\text{AltBit}_1, \text{head}(\text{SQueue}_1))$ and another sequence of $\text{pack}(\text{CheckBit}_2, \text{last}(\text{RQueue}_2))$.

**Embedding The Half-protocol into The Complete Protocol.** The complete protocol is the program $P = (V,I,T,C)$, where:

- $V \equiv \{ \text{AltBit}_1, \text{CheckBit}_1, \text{Channel}_{12}, \text{CanLose}_{12}, \text{SQueue}_1, \text{RQueue}_1, \text{AltBit}_2, \text{CheckBit}_2, \text{Channel}_{21}, \text{CanLose}_{21}, \text{SQueue}_2, \text{RQueue}_2 \}$
- $I \equiv \{ s \mid s(\text{Channel}_{12}) = \emptyset \land s(\text{Channel}_{21}) = \emptyset \land (s(\text{AltBit}_1) = s(\text{CheckBit}_2) \land s(\text{AltBit}_2) = \text{neg}(s(\text{CheckBit}_1)) \lor (s(\text{AltBit}_1) = \text{neg}(s(\text{CheckBit}_2)) \land s(\text{AltBit}_2) = s(\text{CheckBit}_1)) \}$
- $T \equiv \{ t \mid \forall x. (x \in V \land \text{Channel}_{12}, \text{Channel}_{21}, \text{SQueue}_1, \text{SQueue}_2) \Rightarrow \text{DontChange}(t, x) \land \text{MayWrite}(t, \text{SQueue}_1) \land \text{MayWrite}(t, \text{SQueue}_2) \land \text{MayRead}(t, \text{Channel}_{12}) \land \text{MayRead}(t, \text{Channel}_{21}) \}$

We use the results exposed in this paper to immerse $P_1$ into $P$. We do so twice. One time, using the simple function $\varphi$ corresponding to the erasure of
added variables. Another time using the function $\varphi'$ which is similar to $\varphi$ except it swaps indices:

\[
\begin{align*}
\varphi : \mathcal{S}_V &\to \mathcal{S}_V, s \mapsto s_1 \\
\varphi' : \mathcal{S}_V &\to \mathcal{S}_V, s \mapsto s_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>$s_1(\text{Channel}_1) = s(\text{Channel}_1)$</th>
<th>$s_1(\text{Channel}_2) = s(\text{Channel}_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1(\text{AltBit}_1) = s(\text{AltBit}_1)$</td>
<td>$s_1(\text{AltBit}_2) = s(\text{AltBit}_2)$</td>
</tr>
<tr>
<td>$s_1(\text{CheckBit}_2) = s(\text{CheckBit}_2)$</td>
<td>$s_1(\text{CheckBit}_1) = s(\text{CheckBit}_1)$</td>
</tr>
<tr>
<td>$s_1(\text{CanLose}_1) = s(\text{CanLose}_1)$</td>
<td>$s_1(\text{CanLose}_2) = s(\text{CanLose}_2)$</td>
</tr>
<tr>
<td>$s_1(SQueue_2) = s(SQueue_2)$</td>
<td>$s_1(SQueue_1) = s(SQueue_1)$</td>
</tr>
<tr>
<td>$s_1(RQueue_2) = s(RQueue_2)$</td>
<td>$s_1(RQueue_2) = s(RQueue_2)$</td>
</tr>
</tbody>
</table>

Notice that $\varphi'$ is defined using the symmetry of the protocol. These two functions satisfy the four conditions given in part 3. Using these functions, we lift the lemma 1 in two others and we obtain:

**Lemma 2** $u_1 \text{ leadstop } v_1$.

**Lemma 3** $u_2 \text{ leadstop } v_2$.

**The Complete Protocol.** To prove the correctness of this protocol, we first prove the following invariant of the program $P$:

**Lemma 4** \(\text{invariant}_P\) \(\begin{align*}
(\neg s(\text{AltBit}_1) = s(\text{CheckBit}_1) \land \neg s(\text{AltBit}_2) = s(\text{CheckBit}_1)) \\
\land (s(\text{AltBit}_1) = s(\text{AltBit}_2) \land s(\text{CheckBit}_1) = s(\text{CheckBit}_2))
\end{align*}\)

Then, we obtain the theorem 4 by applying the lemma 2 then replacing the invariant\(^2\) given in lemma 4 and finally using lemma 3:

**Theorem 4** Assume $q$ is a queue value, then:

\[
\begin{align*}
\text{s(SQueue}_1) &= q \\
\text{s(AltBit}_1) &= s(\text{CheckBit}_1) \\
\text{s(AltBit}_2) &= s(\text{CheckBit}_2)
\end{align*}
\]

Using this theorem, we can prove the protocol correctness:

**Theorem 5** If a sequence is in $SQueue_1$, it will be written in $RQueue_2$.

This is obtained by induction on the contents of $SQueue_1$, using the theorem 4. The symmetry of the protocol implies the same result between $SQueue_2$ and $RQueue_1$. That can be proved using again an embedding function swapping indices 1 and 2.

\(^2\) We can proceed thanks to the substitution axiom, which is a derived rule of the Coq-Unity system [5].
5 Conclusion

We have proposed a solution to extend program states, and shown how it can be applied through the A.B.P. example.

The solution proposed in this paper is based on a function \( \varphi \) from extended program states to original program states. It only requires that \( \varphi \) satisfy some properties. The generality of the method enlarges its application. So, it allows:

- program states extension as showed in the example.
- program compositions (in fact, the proof of the A.B.P. can be viewed as a composition of two identical half-protocols).
- program superposition: when we embed the half-protocol in the complete protocol, we superpose the complete protocol program on the half-protocol program. This notion is given in [3].
- other special uses (proofs based on symmetry or similarity, etc.).

Using this method we aim to build libraries of basic systems and properties. Because we do not want to define contexts manually anymore, we shall now develop a high level language allowing powerful context descriptions.

Acknowledgements

I would like to thank Christine Paulin, Pierre Crégut and Barbara Heyd for their help in the working out of this paper. I am also grateful to Gwendal Blorec and anonymous referees for their suggestions.

References

Formally Analysed
Dynamic Synthesis of Hardware

Kong Woei Susanto and Tom Melham

Department of Computing Science
University of Glasgow, Glasgow G12 8QQ, Scotland
\{susanto,tfm\}@dcs.gla.ac.uk
http://www.dcs.gla.ac.uk/~tfm/dynhw

Abstract. Dynamic hardware reconfiguration based on run-time system specialization is viable with Xilinx XC6200 series FPGAs. The research challenge for formal verification is to help ensure the correctness of dynamically generated hardware. In this paper, the approach is to verify a specialization synthesis algorithm used to reconfigure FPGA designs. The verification approach is based on a deep embedding of a language for netlists and the relational hardware modeling style.

1 Introduction

Most micro-electronic circuit design is done as ASIC design. The design is validated extensively, either by simulation or by formal verification methods, before it is manufactured. The production of the design as a chip will take several months before the designer can test the hardware itself. If the system fails to satisfy the functional specification at the testing stage, the designers must redesign the circuit and redo the whole validation, production, and test cycle.

The flexibility of implementing a design in a short period has made Field Programmable Gate Arrays (FPGAs) popular for rapid system prototyping. The FPGA is an array of generic function cells configured by assigning configuration bits to specify each cell’s functionality and interconnection with other cells. The fabrication phase of the ASIC design cycle is replaced by configuration of an FPGA chip, which takes at most a few seconds. Furthermore, FPGAs make it possible to implement designs for small production markets.

Most reconfigurable hardware designs use the reconfigurable capability of FPGAs only by swapping in pre-compiled circuits from a library. Research at the University of Glasgow wants to exploit more advanced capabilities of FPGAs to tune circuits to have better performance at run time. The run-time reconfiguration needed for this is viable now with the Xilinx XC6200 series FPGAs, though future generations of FPGA chips are likely to offer similar capabilities. The XC6200 chip has SRAM reconfigurable cells, so that changing the implemented circuit’s functionality is as simple as assigning to a variable in a software system.

Our scheme for run-time synthesis is designed for circuits which have both static and dynamic inputs. Imagine a decryption circuit which has two inputs:
the key and the data to be decrypted. Suppose the key changes much less frequently than the data. When the system gets the key, the decryption circuit can profitably be specialized for that specific key. This is done by simplifying components using knowledge of the key value, and reconfiguring the FPGA accordingly. The system will then have a shorter critical path, and so will have a faster decryption process. The key value is known only at run-time, so the system can be specialized only at run-time as well. Our specialization method is adapted from the existing specialization ideas of run-time partial evaluation for software [7].

To ensure the correctness of the system, we can no longer rely on simulation based techniques, which take a long time to execute. Specialization occurs at run-time, and since the specialized circuitry is used immediately, there is certainly no time for simulation. Instead, formal verification will be used to establish the correctness of the specialization process itself. Our verification method is based on formal specification and proof in higher order logic.

The rest of this paper is structured as follows. A brief sketch of partial evaluation techniques for hardware is given in section 2. In section 3, the design flow for our use of FPGAs and the verification approach for this design flow are described. Section 4 explains our verification methodology. In section 5, the FPGA device model used and the embedding of the HDL language semantics are presented. Finally, our conclusions and some ideas for future work are given in section 6.

2 On-line Dynamic Hardware Synthesis

Partial Evaluation is a common technique in software compilations and executions [7]. The method reduces time resource requirements by exploiting the nature of some system inputs which are static for a long period relative to other inputs. Consider a function $f$ that operates on some data. Suppose the input data can be divided into a static (known data) input $s$ and a dynamic (unknown data) input $d$. A partial evaluator is a function $\text{PE}$ that is applied to the (source code for) the function $f$ and the data value $s$ to yield a residual function $f_s$ (eq 1). Moreover, $f_s$ has the same action on the dynamic data $d$ as the generic function $f$ (eq 2). As the result of partial evaluation, $f_s$ runs much faster for each $d$ than the original function $f$.

\[
\text{PE}(f, s) = f_s
\]

\[
[f_s](d) = [f](s, d)
\]

The basis of our work is the observation that partial evaluation can also be applied to circuit descriptions. Partial evaluation for hardware is done essentially by propagating known input values and specializing the corresponding gates. Similar techniques are known as data unfolding or constant propagation. The idea behind our work is to apply this technique to circuits developed for FPGAs, and to do the specialization at run-time. The specialization occurs by
dynamically modifying the configuration data of Xilinx XC6200 chips. Hardware is different than software in the sense that hardware is normally static rather than dynamic. For example, the typical case of dynamic behavior under partial evaluation in software is unfolding of function calls to generate more program text. By contrast, a hardware circuit is always already fixed. Partial evaluation merely specializes the hardware to make some of the circuit disappear, rather than generating new circuitry.

To illustrate partial evaluation, consider the following full-adder, described at the gate level by a Lava [11] function:

```haskell
type FullAdder = ((bit,bit),bit) -> Out (bit,bit)
fa :: FullAdder
fa ((a,b), carryIn)
  = do partSum <- at (0,0) $ xor2 (a,b)
      sumOut <- at (1,0) $ xor2 (partSum, carryIn)
      carryOut <- at (1,1) $ mux2 (partSum, (a, carryIn))
      return (carryOut, sumOut)
```

The ‘at (x,y)’ and ‘$’ notations mean that the gate written after the ‘$’ is placed on the chip at a relative address indicated by the (x,y) coordinates. Suppose the first input a of the full-adder is known as a static input that will remain the same for many different values on b. The circuit can then be specialized and simplified using this value of a. For example, if the value of a is known to be zero, the XOR gate can be simplified to a buffer. The input signal b can directly be propagated to the next component. The overall system is then simplified to a two input function with two components:

```haskell
specialisedFa (b, carryIn)
  = do sumOut <- at (1,0) $ xor2 (b, carryIn)
      carryOut <- at (1,1) $ mux2 (b, (0, carryIn))
      return (carryOut, sumOut)
```

![Circuit diagram of full-adder (a) and specialized full-adder (b)](image_url)

**Fig. 1.** Circuit diagram of full-adder (a) and specialized full-adder (b)
Figure 1 shows the circuit diagram of both full-adders, the original Lava function description and the specialized one. The specialized full-adder has less delay, because it has one less gate along the critical path.

Another example of the idea is illustrated by the 5 by 6 bit parallel multiplier shown in Fig. 2. The dynamic input registers represented by \( a_0 \) upto \( a_5 \) come from the top of the figure, the static input registers represented by \( b_0 \) upto \( b_4 \) come from the right hand of the figure, and the output registers is on the bottom of the figure represented by \( s_0 \) upto \( s_9 \).

![Fig. 2. The shift add 5 by 6 multiplier circuit](image)

Assume at run time the input \( b \) is going to have the known value of 6 for many iterations. The circuits can then be specialized and reduced to have only one addition operation and several long wires. The \( b \) register can disappear entirely (Fig. 3). The resulting circuit, then, can be operated at a higher speed than the original.

The essential requirement for on-line specialization is that the performance gained by specializing must, over the whole input data, offset the cost of doing the specialization. To illustrate the idea, consider a data stream consisting of some specialization parameters followed by \( n \) data items. There are 5 timing parameters to be considered for the specialized system and generic system schemes. For the specialization scheme, we have 3 parameters: the time to synthesize hardware for specialization \( (T_s) \), the programming time to reconfigure the FPGA \( (T_p) \), and the cycle time to process the data using the specialized circuit \( (T_c) \). The generic system has two parameters: the time to load the specialization parameter \( (T_k) \), and the cycle time to process the data using the
Fig. 3. The specialized 5 by 6 multiplier circuit with the b value of 6

generic circuit \( (T_g) \). The time needed to complete the whole process for both the specialized and generic circuit is presented in equations 3 and 4. A better performance can be achieved either when the number \( n \) of data items to be processed is sufficiently large so that the time needed to specialized the circuit \( (T_s + T_p) \) becomes relatively small or when the cost of specialization \( (T_s + T_p) \) over the whole input data is more efficient than the overall cost by the generic circuit.

\[
\text{Specialized Circuit} = T_s + T_p + nT_c \quad (3)
\]

\[
\text{Generic Circuit} = T_k + nT_g \quad (4)
\]

3 Design Flow and Run-time Partial Evaluation

The Xilinx XC6200 series FPGAs have an SRAM reconfiguration facility which allows them to be dynamically reconfigured in a very short time. The most important feature of this chip is that each cell can be configured individually without having to reconfigure the entire chip. This means that it is possible to reconfigure part of the chip while in other parts of the chip some systems are still running.

A notable feature of FPGA design generally is that placement takes an important role owing to the limited connections between cells available on FPGA chips. Most FPGA system designers therefore already have some kind of placement patterns in mind for their designs. Unfortunately, most Hardware Description Languages (HDLs) do not accommodate this important information. The
Department of Computing Science at Glasgow and the University of Chalmers therefore developed an HDL called Lava which incorporates this information [11]. The Lava language is developed as a library in the Haskell programming language; Haskell itself is a pure functional programming language [6].

![Diagram of Lava Description, Lava, EDIF, XACT, CAL, and PE](image)

**Fig. 4.** The design flows for the partial evaluation scheme

The overall circuit development environment (design flow) for the partial evaluation scheme is shown in Fig. 4. The circuit is developed using the Lava programming language. The Lava hardware description is then synthesized to produced a netlist format (EDIF) of the circuit. By using the Xact tools from Xilinx, the circuit at the netlist level is then placed and routed. This process produces a CAL (Configurable Array Logic) description, which consists of the configuration bits of each individual cell in the chip. All these design processes are done off-line. The on-line / run-time reconfiguration part takes place after the circuit is already placed in the chip and is in operation.

In the run-time part of the process, the circuit description in the form of a CAL file is downloaded into the chip’s SRAM. At this stage, the chip is ready to perform the system’s functionality. When the static input is detected, the partial evaluation program analyses the static value and specializes the circuit already placed on the chip. It immediately reconfigures the circuit by generating new configuration data which specializes the current circuit on the FPGA chip. As the result, the circuit on the chip will be simpler and faster than the generic one. Preliminary results of an experiment for run-time specialization based on a simple constant propagation scheme for partial evaluation are presented in [8].
4 Formal Hardware Verification Approach

The correctness of the overall system depends on the correct functional behavior of the specialized circuit with respect to its generic circuit source for given static values. Normally, the correctness of such a system can be investigated by simulation or by formal verification using equivalence checking or model checking. But in our highly dynamic setting, we may generate hardware and use it for only a few seconds before discarding it. Clearly there is no time for simulation or verification. Our approach is therefore to verify the correctness of the synthesis algorithm for the partial evaluation process. If the correctness of partial evaluation can be assured for any source circuit, then system correctness can be concluded. Of course, the result of partial evaluation will be correct only relative to the correctness of the original design. But this can be checked off-line.

The equation shown below states the correctness criterion in general terms. It is similar to software partial evaluation presented earlier. The specialized circuit will have the same behavior as the generic ones when applied with a specific static value.

\[
[circuit](\text{static, dynamic}) = [\text{PE(circuit, static)}](\text{dynamic})
\]  

Our verification method is as follows. The circuit at the CAL level is specialized by a partial evaluation algorithm written in C++. The result is a new circuit configuration, which our run-time system places on the FPGA. The partial evaluation part of this is formalized and verified in the PVS environment. The generic function unit that lies within each FPGA cell is modeled using the standard relational modeling style in higher order logic. The syntax and semantics of circuits at the netlist level is then embedded using the deep embedding method [2]. The embedding uses configured generic function unit models as the hardware model. The partial evaluation algorithm is then described abstractly as a PVS function and a proof of the correctness theorem above is done. The same process of specialization as is done by the actual C++ coded partial evaluator is carried out by the abstract representation of this algorithm in PVS. This results in a similar transformation of netlists as occurs at the CAL level (Fig. 5). Of course, the relationship between our PVS verification and the actual specialization code is only informal, but we aim to make it close enough to justify at least some confidence in the correctness of the actual code.

Partial evaluation at the CAL configuration level will eventually have two main components: reconfiguration of cell functionality, and routing reconfiguration. Routing reconfiguration is needed to exploit the connectivity resources of the XC6200 for larger speedups. Verification will cover both parts of the specialization process. At present, our work on verification has addressed only transformations of configuration of the generic function unit. The netlist deep embedding is currently under construction. A bit further in the future will come an extension of the verification to cover routing transformations.

The design flow in Fig. 4 includes Lava descriptions of the circuits. Preliminary work on a shallow embedding of the Lava language using functional
Fig. 5. The partial evaluation of circuit at the CAL level and proof of the specialization scheme

hardware modeling highlighted two problems: Lava does not include routing information, and the Lava circuit model abstraction is too far from the CAL level, resulting a vague relation between algorithms at those two level. For these reasons, our work on formal verification is not be targeted at the high level Lava HDL level, but at the EDIF and CAL level.

5 Verification Models for Netlist HDL

5.1 Hardware Modeling

The Xilinx XC6200 series has a unique hierarchical architecture. The FPGA surface is organized as an array of simple cells. The contents of each simple cell is called a function unit (Fig. 6). A function unit has six components, five multiplexers and one D-type register. The function unit of each basic cell can be configured either as a logic function or a register by supplying some configuration bits for the multiplexers. Each cell is connected to its four borders (north, east, south, and west). Basic cells are grouped into 4 x 4 blocks of 16 cells. A 4 x 4 array of these 4 x 4 blocks of cells forms a 16 x 16 block. This hierarchical structure is repeated until 64 x 64 blocks or 256 x 256 blocks are formed, depending on the chip series. The whole structure is then surrounded with I/O pads. A similar scheme for routing appears within the hierarchical structure. A 4 x 4 cell block has its own associated routing resources, which provide fast interconnections. This interconnection capability is known as the length 4 fast wire. A similar routing hierarchical structure appears on the 16 x 16 block, the 64 x 64 block, and the 256 x 256 block. In addition, the FPGA series we use introduces a special fast wire called a magic wire. A magic wire has the capability to route signals from individual cells to certain points on its 4 x 4 block border.
In PVS, the behavior of the function unit of the basic cell is specified using the relational modeling style. The function unit can be developed by using 3 basic components: the inverter, the two input multiplexer, and the D-type flip-flop. All other components in the function unit can be developed by using this three basic components. One important feature in the model is that every component has time varying inputs and outputs. The formal model employs the usual notion of signal, which is a boolean valued function taking discrete time arguments.

The three basic specifications are shown in the PVS theory in Fig. 7. The inverter \( inv \) has two ports, a single input \( i0 \) and output \( o \). The multiplexer \( mux2 \) has four ports, three inputs \( i0 \), \( i1 \), \( sel \) and a single output \( o \). These two component models are based on the zero time delay assumption. The D-type flip-flop model is implemented using a unit time delay. The system samples the input value (\( din \)) when the clock rises (\( rclk \)) and holds the value on the output \( q \) until the next rise of the clock. The second output (\( qnot \)) is an inverted output function of \( q \). A more detail explanation of abstract time modelling can be found in [9].

\[
i0,i1,sel0,o,din,q,qnot,clk,clr: \forall \text{VAR signal[boole]} \\
t: \forall \text{VAR time} \\
inv(i0,o) = \forall t. o(t) = \neg i0(t) \\
mux2(i0,i1,sel0,o) = \forall t. o(t) = (sel0(t) \Rightarrow i1(t) \land i0(t)) \\
rclk(clk,t) = \neg clk(t) \land clk(t+1) \\
rdf(din,clk,clr,q,qnot) = \\
(\forall t. q(t+1) = (clr(t+1) \Rightarrow (rclk(clk,t) \Rightarrow din(t) \land q(t)) \land \text{FALSE}) \land \\
(\forall t. qnot(t) = \neg q(t))
\]

**Fig. 6.** Function unit circuit diagram of the FPGAs XC6200

**Fig. 7.** The basic relational specifications: inverter, two input multiplexer and D-type register
The FPGA function unit relational model is presented in Fig. 8. The formal
implementation description is simply a direct transcription into logic of the
circuit diagram on Fig. 7 (which itself is given in the Xilinx data sheets) [13]. The
variables $zf$ up to $f$ are the external I/O ports of the function unit. The variables
$sy_{2..0}$ up to $es$ are the configuration bits which determine the external behavior
of the function unit. The internal interconnection of the system is hidden by the
standard method of existential quantification. A more detailed explanation of
hardware modeling in PVS is presented in [12].

% external I/O ports
x1,x2,x3,clk,clr,f : VAR signal[bool]
% configuration bits
sy2.0,sy2.1,sy3.0,sy3.1,rp,cs : VAR signal[bool]
funit (x1,x2,x3,clk,clr.f,sy2.0,sy2.1,sy3.0,sy3.1,rp,cs) =
\exists (Y2,Y3,RPM,C,S,buf1,buf2,buf3,qnot).
\forall t. inv(x2,buf1) \land
inv(x3,buf2) \land
inv(qnot,buf3) \land
mux4(x2,qnot,buf1,buf3,0,0,0,Y2) \land
mux4(x3,buf2,buf3,qnot,0,0,0,Y3) \land
muxn2(Y3,Y2,x1,C) \land
muxn2(C,qnot,rp,RPM) \land
fff(RPM,clk,clr,S,qnot) \land
muxn2(S,C,cs,f)

Fig. 8. The XC6200 FPGA function unit model in PVS

As an example of verification using this model, consider an AND gate imple-
mented by setting configuration bits (Fig. 9). The signals $sig_{zero}$ and $sig_{one}$
are the ground and $vcc$ sources respectively. These two signals are used to model
the SRAM configuration bits, which have the same behaviour as signal sources.
Only some of the input configuration bits are needed to configure the function
unit. The unused configuration bits can be ignored by simply existentially quanti-
yfying them. Finally, the configured function unit model can be proved correct
with respect to a high level behavioural specification and2_spec. At present, 14
possible configurations have been verified in our PVS theory.

5.2 Semantics Embedding Approach

In the design flow in Fig. 4, the Lava program synthesizes a structural circuit
description and produces a flattened description of the circuit in the EDIF Ver-
ion 2.0.0 syntax. The netlist describes the components as simple gates and
models the interconnection between components. In our verification, circuits at
the netlist level will be specified in higher order logic notation using the deep
embedding methodology. As already mentioned, the choice of a deep embedding
and2\((in1, in2, out1) = \\
\exists (buf2, buf3, sg0, sg1, clk1, clr1).
\forall t. \text{sig\textunderscore zero}(sg0) \land \\
\text{sig\textunderscore one}(sg1) \land \\
\text{funt}(in1, in2, in1, clk1, clr1, buf3, sg0, sg1, sg0, buf2, sg1) \land \\
\text{inv}(buf3, out1)
\)

\text{and2\_spec}(in1, in2, out1) = \forall t. out1(t) = in1(t) \land in2(t)

\text{tm\_and2}: \text{THEROREM and2\((in1, in2, out1) = and2\_spec(in1, in2, out1)\)

\textbf{Fig. 9.} Functional behaviour modeling based on the function unit and proof

of netlists was driven by previous experienced with a shallow embedding of the Lava HDL semantics. The shallow embedding limited the proofs to functional properties. Furthermore, the vague correspondence between the HDL description and what actually happen on the chip gave a less useful verification result.

The netlist syntax we will embed in PVS is presented in Fig. 10. A netlist description contains a cell library which consists of a collection of cells. The circuit as a whole is also part of the library and is a cell constructed from the predefined basic cells in the library. The Lava netlist generator generates circuits as a single flattened cell. Within the cell, the circuit is described as components and their interconnection relations. This structure at the netlist level is maintained in the higher order logic hardware model. The netlist syntax follows the standard EDIF format, which makes the language well structured. The deep embedding semantics will be implemented as a function which takes a circuit in the netlist form as its argument and produces a circuit in the relational form as a result.

cell\_library ::= library\_name cell\_name [interface]* [instances]* [net]*?
interface ::= interface\_name [direction]
direction ::= INPUT | OUTPUT | INOUT
instance ::= instance\_name cell\_name cell\_location
net ::= net\_name [pin]^*
pin ::= interface\_name instance\_name
library\_name, cell\_name, cell\_location, interface\_name,
instance\_name, net\_name ::= identifier
identifier ::= [A-Za-z][A-Za-z0-9]*

\textbf{Fig. 10.} Simplified netlist syntax from Lava netlist generator. The optional items are followed by '\?', and the repeated items are followed by '\*'.

The run-time partial evaluation algorithm is implemented in the C++ language. The algorithm uses a constant propagation scheme by simply propagated
C++:
if (fn == INV)
  { if (input == cell[i][j].input_a)
        { if (value_a == ZERO)
          { cell[i][j].cell_function = ONE;
            setFunction(i, j, ONE, cell[i][j].input_a);
            return ONE;}}}}

PVS:
test_inv_low : THEOREM cell_inv(0) = cell_high

Fig. 11. Transformation modelling from C++ to PVS

the static value and specializing all the corresponding cells into a simpler cell. Consider an example, an inv gate cell. If the input of the inverter has a static value of 0, then the cell can be specialized to a src source (Fig. 11). All the possible cell transformations have been implemented in PVS and proved correct (52 cell transformations have been verified). The next step is to develop a high level abstraction of the algorithm which captures all possible circuit netlist transformations in PVS. We will then prove that the partial evaluation function applied to the circuit netlist and the static values will result a specialized circuit netlist whose semantics agrees with that of the original (eq 5).

6 Summary and Future Work

Run-time circuit specialization poses an interesting challenge for verification of the specialization algorithm. Our approach is based on two aspects: hardware component modeling, and semantic embedding of the circuit description language. The problem is addressed in two hierarchical steps, which reflect the design cycle: proving the system correctness at the netlist level and then extending the system with the routing information present in the CAL level. The hardware model at the netlist level is based on a generic function-unit. This function-unit can be configured to perform a certain functional behavior, either to be as a logic function or a register. At the current stages of our work, 14 configurations of basic cell functionality and 52 cell transformations have been verified. The model then will be used in verifying the specialization transformation algorithm based on a high level abstraction of the implemented C++ algorithm. We will do this by developing a deep embedding of a netlist semantics and defining a high level specialization algorithm in the PVS environment.

Acknowledgments

This work is part of a project funded by UK-EPSRC (project reference GR/L 38530), the United Kingdom Ministry of Defence, and Xilinx Inc. It is managed by Tom Melham (University of Glasgow), Satnam Singh (Xilinx Inc.), and
Derek McAuley (Microsoft Research Labs, Cambridge). The run-time partial
evaluation system is developed by Nicholas McKay.

The authors thank the SRI team for making the PVS system available to
perform the research presented in this paper. We also thank Richard Boulton
for providing us with an unpublished paper on embedding netlists in higher order
logic [1], and Paul Jackson for valuable help with PVS.

References

2. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and
   John Van Tassel. Experience with embedding hardware description languages in
   HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, Theorem Provers in Circuit
   Design: Theory, Practice and Experience: Proceedings of the IFIP WG10.2
3. Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A dyna-
   mic reconfiguration run-time system. In IEEE Symposium on FPGAs for Custom
   Computing Machines: Preliminary Proceedings, Napa Valley, California, pages
4. Charles Consel and Oliver Danvy. Tutorial notes on partial evaluation. In ACM
5. Patrick W. Foulk. Data-folding in SRAM configurable FPGAs. In IEEE Sym-
   posium on FPGAs for Custom Computing Machines, Napa, California. pages 163–
   171. April, 1993.
   Report YALEU/DCS/RR-901, Yale University, May 1996.
8. Nicholas McKay and Satnam Singh. Dynamic specialisation of XC6200 FPGAs
   by partial evaluation. To appear in Proceedings of Field Programmable Logic and
9. T. F. Melham. Higher Order Logic and Hardware Verification. Cambridge Univer-
10. Satnam Singh, Jonathan Hogg, and Derek McAuley. Expressing dynamic recon-
    figuration by partial evaluation. In Kenneth L. Pooek and Jeffrey Arnold (eds.), IEEE
    Symposium on FPGAs for Custom Computing Machines, Napa Valley, California,
11. Satnam Singh and Mary Sheeran. Designing FPGA circuits in Lava, unpublished
    paper.
12. Madayan Srivas, Harald Rueß, and David Cyrluk. Hardware verification using
    PVS. In Thomas Kroft, editor, Formal Hardware Verification Methods and Systems in
Requirements for a Simple Proof Checker

Geoffrey Watson

Software Verification Research Centre
University of Queensland, 4072, Australia
gwat@csee.uq.edu.au

Abstract. This paper discusses some of the requirements for a stand-alone proof checker. It discusses the requirement on the theorem prover to provide a record of the proof suitable for checking, and it focuses on the advantages obtained when the theorem prover cooperates with the checker by supplementing and formatting the proof record.

1 Introduction

Ensuring confidence in machine generated proofs is an important issue in automated theorem proving. One approach is to pass a record of the proof to an independent proof checker for validation. This is proposed, for instance, as a means of overcoming a lack of confidence in machine generated proofs in mathematics [14].

The background to the current work is the use of theorem provers as support tools in software verification using formal methods. The balance of costs against benefits makes the use of formal methods most attractive for the development of high integrity software. In such applications it is especially important that the results be free from error. Hence the recommendations for the use of a separate proof checker found in standards for the development of high integrity systems such as the British Ministry of Defence Standard 00-55 [1].

The rationale for using a proof checker is that the computation of the validity of the proof is performed by by two independent pieces of software. Our confidence in this process would be increased further if one or both of the programs were formally verified, and hence ‘guaranteed’ to be correct. Since a proof checker is inherently simpler than the theorem prover whose proofs it checks, it is the obvious choice for formal verification. Formal verification is still a complex and lengthy activity, so it is important that programs to be verified be as simple as possible. This paper discusses some of the issues pertinent to ensuring that a proof checker, as a candidate for formal verification, is kept simple. The primary focus is on the issue of the cooperation of the theorem prover with the proof checker.

The rest of this paper is organised as follows: Sect. 2 reviews how proofs are currently represented by theorem provers and Sect. 3 discusses the recording of proofs for use by proof checkers. Section 4 looks at approaches to designing proof checkers that are simple and therefore feasible to verify and discusses the principles of cooperation between prover and checker. Section 5 discusses future work.
2 The Representation of Proofs by Theorem Provers

The input to a proof checker is a representation of a proof constructed by a theorem prover. A survey of current theorem provers [15] revealed that few of them support the export of proofs in a format suitable for independent checking. In most cases a proof can only be saved in a form suitable for reuse by the prover itself. Usually this is in the form of a script of commands that can be replayed by the prover to regenerate the proof. In this section we briefly review a few of the more prominent current theorem provers from the perspective of how they represent and record proofs.

Otter [7] is a semi-automatic resolution theorem prover used for the investigation of mathematical problems. The user specifies the theorem to be proved and the axioms to be used. They may also specify how the proof method is to be applied, for example the searching strategies and inference rules to be used. Otter may be instructed to print out the proof on completion, either as a list of the inference rules that were actually used in the proof, or as a Lisp-like representation of the proof. This latter format is designed for submission to an external proof checker for certification and it contains all the information required for the proof explicitly. This proof format, and a checker for Otter proofs written in Nuprl, have been used by McCune in validating the solution of the Robbins problem [8].

Mizar [12] is a checker for mathematical proofs. It has a language for developing and recording formalised mathematics, and the system has extensive software support including a proof checker. The Mizar philosophy is that proof steps should be obvious [13], and the checker is intended to accept such obvious steps and reject all others. A Mizar user writes a detailed proof of the theorem in the formal Mizar language. This is then submitted to the checker. If it fails the user has to modify the script – either by correcting errors, specifying the proof in more detail, or perhaps by decomposing the proof and proving preliminary lemmas first. The record of a Mizar proof is just the input script.

In the area of program verification, three of the most successful theorem provers are PVS, HOL and Isabelle. PVS [9] is a proprietary system for developing formal specifications. In the PVS proof checker the emphasis is on powerful automated procedures that hide irrelevant detail from the user. The proof is conducted top down, being broken into a hierarchy of subsidiary goals that are proven in turn. PVS provides the user with a literate explanation of each step so that the progress of the proof is easily understood. However the record of the proof stored externally is as a Lisp-like expression which represents the commands used to construct the proof, that is in a form of script useful to PVS itself.

Standard HOL [4] also does not represent proofs explicitly, however there are extensions to HOL that do so:

  The proofs are recorded and can then be translated to a literate format for reading by human beings.
- Wong [16] describes a method for recording HOL proofs to facilitate the external checking of proofs. This was part of a larger project [3] that included the formal specification and abstract development of a HOL proof checker. Both systems are implemented by extensions to the standard HOL system, and in both cases the intended mode of use is to record the proof by rerunning a previously completed proof with recording switched on.

Isabelle [10] is a generic theorem prover. In normal use Isabelle does not record proofs in detail. However recent versions of Isabelle give the user the option of recording the proof [11]. The proof is stored as a detailed record that can be exported and is intended for proof checking. However the recorded format is not explicitly documented and the Isabelle User Manual discourages its use for proof checking.

Implementations of proof checkers are few in comparison to theorem provers. Exceptions include the checker used by William McCune to verify the EQL/Otter solution to the Robbins problem [8] and Wai Wong’s proof checker for HOL [16] (both mentioned above) and also the Eves checker [6]. Actual implementations tend to be highly specific to the particular theorem prover for which they are written.

3 Recording Proofs for Checking

The brief survey of proof representations by theorem provers in Section 2 has indicated that in general the external representation of their proofs is in the form of a high-level script. This conclusion is supported by the results of a larger survey [15]. A high level script is not the most suitable record for checking the proof (rather than re-proving the theorem from scratch). Therefore, in order to implement a proof checker, we may also have to extend the theorem prover to record and export proofs in a more suitable format.

Even when a theorem prover maintains a detailed record of the proof during its construction, this often only records what is left to be proved, rather than a full record of the proof so far. On completion, such a strategy will have reduced the proof record to just ‘true’. The lack of external recording of proofs is commonest among provers used for formal software development, perhaps because in such a context validity is the main focus. In contrast, in mathematical contexts the proofs themselves are of interest in their own right.

Emitting a sequential external record of the proof during its construction (rather than constructing an internal representation of the proof and using this to output a record of the proof on its completion) is not in general a suitable strategy. This is not least because, in the case of interactive provers, it would record all the false starts and errors made by the user. Although this problem can be overcome by undertaking proof recording as a separate activity during a rerun of the completed, and optimised, proof, other considerations (described in Sect. 4) make the derivation of the proof record from a complete internal representation of the proof more desirable.
The export of a proof for checking may be viewed conceptually as a separate activity from the recording of a proof for other purposes during its construction, although it may be implemented as an integral part of the latter process. The exported proof object is a justification or certificate of the validity of the theorem, whose main purpose is the auditing of the proof process. Literate proof records, such as provided by PVS or Cohn’s HOL proof accounts, have a similar purpose in providing a humanly checkable audit trail of the proof. In this paper however we are considering auditing by machine.

4 Simplifying Proof Checking

Our confidence in the proof checking process will be increased if we believe that the checker is itself correct. Thus we require a simple checker whose specification and code can be thoroughly inspected and tested to give a high degree of assurance in its correctness. Ideally we would like the checker to be developed from a formal specification using formal methods.

One way to ensure that the checker is as simple as possible is to optimise the proof record exported by the theorem prover for its intended purpose: checking by an external tool. The type of record that the theorem prover itself will find useful for recreating the proof may be very different from that useful to an external tool that simply wants to check the validity of the proof.

We propose some simple and self-evident guidelines for the construction of proof records for exporting to a proof checker. The basic premise is that the prover should help the checker wherever possible. This may be termed the principle of cooperation, although the cooperation is one-way, from prover to checker. One consequence of this is that the proof record should contain no unnecessary information which would make the task of the checker harder – we may call this the requirement of relevance. For instance, automatic theorem provers often work by searching large sets of clauses for a refutation, in such cases they should only record the actual branches used to construct the counter example. Conversely another consequence is that the checker should be provided with all information useful to its task, this may be called the requirement of full disclosure.

Finding a proof is often a complex process, it may have involved much searching and possibly the refinement of a vague initial proof sketch. However, when the proof is finished, any ambiguities that may have existed during the search for the proof have been resolved and the proof object passed to the checker should reflect this. Although this may be seen as just an aspect of the requirement for full disclosure, it can also be seen as a means of achieving disclosure in the simplest possible way. The principle here is that the proof should be communicated as a complete and unified object.

This principle entails, for instance, that all rule applications should be fully instantiated, and the parameters be available directly at the point that the rule is used. As an example, the strategy for a goal-directed proof may use the rule:

\[
\frac{A, A \Rightarrow B}{B}
\]
to split the goal $B$ into the two subgoals $A$ and $A \Rightarrow B$. However the exact value of $A$ that is required may not be known until details in some other part of the proof are resolved. It is a common technique for $A$ to be left as a metavariable, to be instantiated later. Such a technique is very useful in improving the usability of goal-directed proof. (This is to be distinguished from the case where we are proving a schematic theorem which contains $A$ as a metavariable.)

A naive method of constructing a proof object in this case would simply record the instance of the rule at the moment it was applied, thus including a reference to the metavariable $A$. However, when the proof is complete the required value for $A$ will have been determined, and a cooperative theorem prover will ensure that this instantiation is applied everywhere in the proof object, including in the initial application of the rule. Such temporary metavariables should never appear in the final proof record, which should only record the required instantiations.

Of course with a prover written in a language such as Prolog, global instantiation may be automatic. But even in this case care may be necessary to ensure that, for instance, the encoding of the proof step into a text format is not performed before the instantiation takes place.

In respect of the requirement of relevance mentioned earlier, note that in this case the proof object does not record how the proof was constructed, but only the completed proof. This is an instance of the general rule that the proof checker should be protected from irrelevant details, and details of the construction process are not relevant to the validity or otherwise of the proof. Indeed the required value of $A$ could have been supplied initially, either by a particularly prescient user or by one who has already completed the proof either on paper or in a previous session.

Often the same proof could have been created by forwards or backwards proof, but the proof object produced may be identical and the proof checker need not know how the proof was obtained. Of course the structure of the proof object may reveal this, for instance whether the inference rules are used forwards or backwards. But it is not necessary that the proof object generated reflect the method used to create it, for instance HOL proofs are recorded as a forwards sequence of primitive inferences even when the proof is performed backwards by the subgoal package.

Full disclosure requires that the prover ensure that all useful or relevant information is present. In order to simplify the task of the proof checker as much as possible, this requirement should extend to auxiliary information that reduces the work to be done by the checker. A prime example of this is that wherever the prover conducts a search it should pass the result to the checker, obviating the need for the prover to duplicate the search. In many theorem provers searching is a major activity and none of this need be performed by the proof checker. For instance a tactic may search for a rule to match the current subgoal. In such cases a proof script will typically just record the tactic as specified by the user. However a proof record for checking should record the result of the search, that is the rule that was found rather than the tactic used to find it.
Another case where the prover can help the checker by providing auxiliary information is where the proof construction requires the solution of a unification. Unification is harder than substitution, so whenever the prover calculates a unifier it should record the resulting substitution for the checker to use - thus simplifying the task of the checker.

These requirements are interrelated. For instance, in many cases unifications occur because of the use of temporary metavariables during proof, so by ensuring full instantiation of temporary metavariables these unifications will be reduced to at most pattern matching in the completed proof object.

5 Discussion and Future Work

We have described some requirements for the input to a simple proof checker. We have also noted that few proof checkers output a proof record in a suitable form. In order to implement a proof checker we therefore need to extend existing provers so that they generate proofs in the form that we require. This can be done either by modifying the prover to record proofs (as in Wong’s extension to HOL) or perhaps by passing a verbose proof record, such as that produced by Isabelle, through a filter to convert them a suitable format.

The Eves [5] system provides a proof checking facility similar to that suggested here and motivated by similar considerations to those described in Sect. 1. Eves uses the term proof log for the external record of the proof. The Eves report on proof checking [6] points out that provided the checker is sound, the generation of the proof log itself is not a high-integrity operation. Errors in producing the log will reduce the efficiency but not the integrity of the system, that is, it may result in correct proofs being rejected but it will not allow incorrect proofs to be validated.

Indeed, in a context where we require a permanent record guaranteeing our theorem proving activities (for auditing or other purposes), we may consider that an interactive theorem proving system deliver two proofs: the proof constructed by and convincing to the original user, which is confirmed by the theorem prover, and a machine auditable proof record that can be checked independently by a proof checker. The guarantee of the validity of the latter is given by the checking process, not by showing its faithfulness to the ‘original’ proof.

The implications of this line of enquiry, where the proof record may be generated by a process that transforms the ‘raw’ proof record in arbitrary ways, before it is passed to the actual proof checker is twofold. Firstly the transformation process can tailor the proof record to suit the proof checker and thus simplify the latter as much as possible, as has been discussed in this paper. One motivation for this is to make the checker amenable to full formal development. The second implication is that it suggests a method of approaching the development of a generic proof checker – a trusted program that can process proofs from a variety of different theorem provers. In this approach the difficulty in mapping a variety of theorem prover proof records into a single generic checker need not impact on the checker itself, rather on the filters or transformers involved.
Acknowledgements

I would like to thank my past and present supervisors: Dr Owen Traynor, Professor Jim Welsh and Dr Peter Robinson for their discussions and insights into theorem provers and proof checking. Geoffrey Watson is supported by an Australian Postgraduate Award.

References

Integrating HOL and RAISE: 
a practitioner’s approach

Wai Wong¹ and Karl R. P. H. Leung²

¹ Hong Kong Baptist University, Hong Kong 
wwong@comp.hkbu.edu.hk

² Hong Kong Polytechnic University, Hong Kong 
cskleung@comp.polyu.edu.hk

Abstract. HOL is strong in theorem proving. This strength can be applied to formal verification in software development. RAISE is a rigorous software engineering method. Although RSL, the RAISE Specification Language, is a formal language, it is not focused on theorem proving. In this paper, the incorporation of HOL and RAISE in software development is discussed. This incorporation provide a pragmatic approach for software engineering practitioners to use formal methods for developing quality software in their daily work.

1 Introduction

RAISE [3] is a rigorous software engineering method developed by industry for developing quality software through step-wise refinement, separate development and rigorous approaches. Although RSL [4], the RAISE Specification Language, is a formal language, it is not as focused as HOL in theorem proving.

On the other hand, HOL [3] has been developed with the objective of proving theorems. In software developments, this theorem proving capability can be applied in formal verifications. These include verifying the correctness of design and implementation against formal specifications.

This paper describes how HOL and RAISE can be incorporated. The objective of the incorporation is to complement these two formal techniques with each other so that it will enhance and enrich the support to software development. The main aim of this incorporation is to enable the use of the theorem proving capability of HOL in verifying specifications and implementations written in RAISE.

2 Background

2.1 HOL

HOL is an interactive theorem proving environment supporting the HOL logic, a version of higher order typed λ-calculus. The HOL system provides a number of facilities for the user to perform proofs in the HOL logic. Expressions of the HOL
logic are terms. There are four kinds of primitive terms: variables, constants, abstractions and combinations. Table 1 lists these primitives. The HOL system also has a set of pre-defined terms for commonly used standard logical operators and quantifiers which are listed in Table 2.

**Table 1.** HOL primitive terms

<table>
<thead>
<tr>
<th>Kind of term</th>
<th>HOL notation</th>
<th>Standard notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>v : ty</td>
<td>( \tilde{v} )</td>
<td>variable of type ( \sigma )</td>
</tr>
<tr>
<td>Constant</td>
<td>c : ty</td>
<td>( \tilde{c} )</td>
<td>constant of type ( \sigma )</td>
</tr>
<tr>
<td>Abstraction</td>
<td>( \lambda x. t )</td>
<td>( \lambda ) ( x ). ( t )</td>
<td>( \lambda )-abstraction</td>
</tr>
<tr>
<td>Combination</td>
<td>( t_1 t_2 )</td>
<td>( t_1 t_2 )</td>
<td>apply function ( t_1 ) to argument ( t_2 )</td>
</tr>
</tbody>
</table>

**Table 2.** HOL non-primitive terms

<table>
<thead>
<tr>
<th>Kind of term</th>
<th>HOL notation</th>
<th>Standard notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>( \neg t )</td>
<td>( \neg t )</td>
</tr>
<tr>
<td>Disjunction</td>
<td>( t_1 \lor t_2 )</td>
<td>( t_1 \lor t_2 )</td>
</tr>
<tr>
<td>Conjunction</td>
<td>( t_1 \land t_2 )</td>
<td>( t_1 \land t_2 )</td>
</tr>
<tr>
<td>Implication</td>
<td>( t_1 \Rightarrow t_2 )</td>
<td>( t_1 \Rightarrow t_2 )</td>
</tr>
<tr>
<td>Equality</td>
<td>( t_1 = t_2 )</td>
<td>( t_1 = t_2 )</td>
</tr>
<tr>
<td>( \forall )-quantification</td>
<td>( \forall x. t )</td>
<td>( \forall x. t )</td>
</tr>
<tr>
<td>( \exists )-quantification</td>
<td>( \exists x. t )</td>
<td>( \exists x. t )</td>
</tr>
<tr>
<td>e-term</td>
<td>( e x. t )</td>
<td>( e x. t )</td>
</tr>
</tbody>
</table>

Every HOL term is associated with a type. The HOL logic allows polymorphic types. A type can be either a type variable which may be instantiated when needed, or a constant type such as : bool (boolean), : num (natural numbers), or a type operator, such as list, pair, and so on. The arity of a type operator is the number of arguments it can take, for example, the arity of list is 1 and the arity of pair is 2. In fact, type constants are just type operators with zero arity. The HOL system has a small set of pre-defined types. It also has some functions to define new types.

### 2.2 RAISE

RAISE is an acronym for Rigorous Approach to Industrial Software Engineering [5]. It was an industrial R&D project carried out in the context of the ESPRIT
I program 315. It is a formal method aimed at constructing software with better reliability, fewer errors, better documentation and easier to maintain. It also aimed at developing a set of notations, techniques and tools that would enable industrial usage of formal methods in software developments.

RAISE consists of the RAISE Specification Language (RSL) and a comprehensive development method. The RAISE tool supports producing specifications, theories and proofs. RSL and the RAISE tool focus on supporting the specification, design and implementation stages of software development process.

The RAISE Method. The RAISE Method [5] is a comprehensive software engineering method. The RAISE method has a set of guidelines for most software development activities.

The RAISE method is based on the stepwise refinement paradigm. Software is constructed through a series of steps. Each step is a refinement of the previous one. RAISE uses the 'invent-and-verify' philosophy in advancing in each step of refinement. A step is first designed 'manually'. Refinement relations can subsequently be verified formally with theory extension [5]. The RAISE tool supports this verification. This is also a sound basis for obtaining correct software. Besides formal proof, verification can be done less formally, i.e. rigorously, if this satisfies the practices and constraints associated with the project.

The RAISE method also supports separate development. A system is divided into modules. Each module can be developed separately and independently. Following the refinement requirements, all modules will preserve the properties of its initial specifications. Hence the final system is then the integration of all these developed modules. These are illustrated in Fig. 1.

In Fig. 1, module $A$ is used in the development of module $B$. Without taking account of the development of $A$ from $A_1$ to $A_m$, $B$ uses $A_1$ in the development from $R_0$ to $B_n$. The notion of separate development ensures that $A_1$ can be substituted by $A_m$ and correctness is preserved. Then, in integrating $B_n$ with $A$, $A_1$ is substituted by $A_m$ to form $B_{n+1}$.

The RAISE method is not prescriptive nor cook-book like. It provides a framework and guidelines for the software development process. This makes the RAISE method adaptable to different techniques and different levels of rigour required in different projects.

There are four major procedures in the RAISE method. They are Specification, Development, Justification, and Translation. Specifications in all stages of software development are supported. These include initial abstract applicative specifications in the requirement engineering stage to concrete imperative or concurrent specifications in the implementation stage. Development procedure is the process of developing more concrete modules. Justification is the process of verifying the correctness of modules developed. Translation is the process of translating RSL specifications into program codes. With the support of the RAISE tool, Ada or C++ programs can be generated from a subset of RSL specifications.
2.3 The RAISE Method and HOL

RAISE is a comprehensive software engineering method which covers all phases of software development from Requirement Specifications to Maintenance. In contrast, HOL is primarily a tool and environment for theorem proving. It has been used in a variety of different areas including formal verification of hardware and software. Because HOL is an open and extensible system, many formalisms have been embedded into it. Because of these embeddings and extensions, HOL has now equipped with a large library of theories and tools that can be used in many different application areas.

Depending on the project requirements and standards, the developer may not need to carry out complete formal reasoning. For example, one can use an informal argument to describe why certain property is believed to be true. When a project requires higher degree of assurance of the correctness, one can carry out a complete formal verification. With its large number of libraries, HOL is very suitable for supporting this formal verification.

3 Integrating HOL and RAISE

RAISE supports most of the tasks in different software development processes. In comparing HOL with RAISE, RAISE is a more comprehensive software en-
gineering method. Since HOL is strong in theorem proving, HOL can play the
role of proving the correctness of design.

RAISE and HOL can be incorporated by the two of the principal concepts
of RAISE. They are Stepwise development and Separate development. Separate
development allows RSL modules to be ‘contract out’ to HOL. Stepwise develop-
ment ensures that the ‘return’ modules from HOL are a valid refinement of
the previous RSL modules.

Let us illustrate this notion with a hypothetical example. Let the RSL mod-
ule \( A_1 \) in Fig. 1 be a module which needs to be proved correct. By separate
development, \( A_1 \) is separated from \( B_0 \). Instead of carrying out justification of
\( A_1 \) in RSL, it is translated into HOL. The HOL version of \( A_1 \) is proved in the
HOL system. After the proof, the resulting HOL module is translated back to
RSL. This module is regarded as the \( A_m \) under the condition that this \( A_m \)
satisfy the two requirements of refinement, i.e.

1. the signatures of \( A_m \) include those signatures of \( A_1 \) and
2. the features of \( A_1 \) are preserved by \( A_m \).

Afterwards, by the notion of separate development, \( A_m \) is a valid module which
can be integrated with the original module \( B_0 \).

The main problems of incorporating RAISE and HOL are the translations
between RSL and HOL. These translations have to preserve the meaning of the
modules.

4 RSL and the HOL Logic

The RAISE Specification Language, RSL, is a wide-spectrum language [4]. It
is inspired by and unifies features of several specification languages including
VDM [9,2], CSP [8] and CLEAR [1]. It supports many paradigms of specifi-
cations, such as abstract, applicative, concrete, imperative and concurrency spec-
fications. Therefore, it is suitable for writing specifications from initial stage to
implementation stage. It has well-formed syntax, well-defined semantics [7,11]
and a proof system [10]. RSL also supports modularisation, parameterisation,
class, polymorphism and inheritance.

In the following description of the features of RSL, we will use a small module-
specification specifying a metric space as an example. This is shown in Fig. 2.

4.1 Modules and Scheme Declaration

Specifications in RSL are organised into modules. The basic structure represent-
ing a module is a scheme declaration. Each scheme usually contains three parts:
type, value and axiom, but all of them are optional. The type part contains
declarations of new types and sorts. The value part contains declarations of new identifiers and their signatures. The axiom part contains assertions of the
properties of values declared in the value part. The scope of the declarations in
a scheme is limited within the scheme, and possibly within its extension.
scheme
   SPACE =
   class
      type
         Point,
      Space : Point-set
   value
      distance : Point × Point → Real
   axiom
      ∀p, p', p'' : Point • distance(p, p) = 0 ∧
      (p ≠ p' ⇒ distance(p, p') ≥ 0.0) ∧
      distance(p, p') = distance(p', p) ∧
      distance(p, p') ≤ distance(p, p'') + distance(p'', p''
end

Fig. 2. A simple RAISE scheme declaration

The HOL logic is effectively flat, i.e., everything loaded in the logic is visible. The theory hierarchy is on the meta-language level for organising the types, definitions and theorems into more manageable units. The theory structure in the HOL base system does not allow theories to be parameterised.

With this understanding, we can map an RSL scheme into a HOL theory. The SPACE scheme in RSL can be translated into a HOL theory with equivalent meaning as shown in Fig. 3. In this theory, a theorem asserting the existence of a DISTANCE function having the required properties is proved, then a constant specification is used to define the constant DISTANCE. The abstract types Point and Space are represented in HOL using type variables. This will be further described in Sect. 4.2.

The RSL scheme declaration introduces an identifier which can then be used to refer to the module. For example, SPACE is the name declared in Fig. 2. A scheme can have parameters which should be declared in a list enclosed by parentheses after the scheme name. This provides a means of parameterise a module. The parameters are objects satisfying a class specification. For example, the SPACE scheme in Fig. 2 could have taken, as its parameter, an object P which is a class containing at least a type Point :

scheme
   SPACE (P : class type Point end) =
   class
      type Space : PPoint-set
      ...

where PPoint refers to the type Point in the class P and -set is the set type constructor.

RSL allows a module to be extended with more declarations. This gives a means of decomposite specifications into comprehensible and reusable units. For
new_theory "space";

val DIST_EXISTS =
  let val p = (--'p':Point'--) and p' = (--'p':Point'--) and 
p'' = (--'p''':Point'--)
in
  prove 
  ( ((--'f. !'p 'p' 'p'. (f("p", "p") = (0 & 0)) \/
      (\'(p = "p'\') \=> ((0 & 0) \<= f("p", "p'"))) \/
      (f("p","p'") \<= f("p", "p'")) \+) \-
      (f("p', "p'') \<= f("p', "p'"))')',--)),
EXISTS_TAC (--'\(@(p',"p'"), & 0)'--) THEN REPEAT STRIP_TAC THEN
CONV_TAC (DEPTH_CONV pairlib.PBETA_CONV) THEN
REWRITE_TAC (map (thm "REAL") ["REAL_LE_REFL", "REAL_ADD_RID"])
end;

val DISTANCE_DEF = new_specification
{consts = [(const_name = "DISTANCE", fixity = Prefix)],
  name = "DISTANCE_DEF", sat_thm = DIST_EXISTS};

close_theory();
export_theory();

Fig. 3. A HOL translation of the simple RAISE scheme

every example, we can extend the SPACE scheme to create a SET_SPACE which
models spaces as a set of points.

scheme
  SET_SPACE =
    extend SPACE with
    class
      ...

Due to the flatness of the HOL logic, the RSL module hierarchy should be
handled at the meta-language level in HOL, i.e., using the HOL theory hierarchy.
For the above example, a new theory SET_SPACE can be created which has the
theory SPACE as its parent.

4.2 Types

RSL has a number of built-in types which can be divided into two kinds:

- Atomic types which are Bool (booleans), Int (integers), Nat (natural num-
  bers), Real (real numbers), Char (characters), Text (character strings) and
  Unit (the singleton type).
– *Composite types* which are \( \times \) (products), \( \to \) (functions), -set (sets), -list (lists), \( \rightarrow \) (maps)

Atomic types and composite types are collectively called *concrete* types.

In the HOL base system, only very small number of types and type operators are defined, such as boolen natural numbers, pairs and lists. However, many more types and type operators are defined in the system libraries, such as the set library. All RSL built-in atomic types have correspondence either in the HOL base system or in a HOL system library. Except \( \rightarrow \) (maps), RSL’s composite types have correspondence in the HOL base system and the system libraries. It is not difficult to develop a map theory in HOL to represent RSL’s map type. Table 3 lists the type correspondence between RAISE and HOL.

**Table 3. Equivalent types between HOL and RSL**

<table>
<thead>
<tr>
<th>RSL</th>
<th>HOL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>one</td>
<td>the type having a single value</td>
</tr>
<tr>
<td>Bool</td>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>Int</td>
<td>integer</td>
<td>integers</td>
</tr>
<tr>
<td>Nat</td>
<td>num</td>
<td>naturals</td>
</tr>
<tr>
<td>Real</td>
<td>real</td>
<td>reals</td>
</tr>
<tr>
<td>(\times)</td>
<td>#</td>
<td>pairs</td>
</tr>
<tr>
<td>(\to)</td>
<td>(\rightarrow)</td>
<td>function</td>
</tr>
<tr>
<td>-list</td>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>-set</td>
<td>set</td>
<td>set</td>
</tr>
</tbody>
</table>

**Abstract Types.** RSL allows users to defined *abstract* types in the *type* part of a *class* declaration. For example, the Point type declared in Fig. 2 is an abstract type. The values contained in an abstract type is not specified. However, one can specify functions on abstract types by give them certain properties. For example, we do not know what are the values in the type Point, but, we know that the function distance will measure the distance between two points and the distance is a real number.

RSL’s abstract types are usually used to reflect the requirements where no information is given. This allows *under-specifications* to be written. After some refinement steps, the abstract type can be developed into a concrete type by giving it a direct definition as some type expression, e.g., Point = (Real \(\times\) Real).

RSL’s abstract types can be represented in HOL by type variables since we known nothing about the type itself. When an RSL abstract type is developed into concrete type, the corresponding HOL type variable can be instantiated with appropriate types.
**Subtypes.** RSL also supports *subtypes*. A subtype is defined by constraining a type with a predicate. For example, the subtype expression \{r : \textbf{Real} \land r \geq 0\} represents a subtype of \textbf{Real} which contains all non-negative real numbers. The distance function declared in the SPACE scheme could have returned a value in this subtype.

The HOL base system does not have subtypes. However, restricted quantifiers can be used to simulate subtypes. For example, a variable that can stand for any non-negative real numbers can be restricted in the following form:

\[ \forall r : \text{real} :: (\lambda x. r \geq 0). \ldots \]

### 4.3 Value Declarations and Definitions

The *value* part of a module contains declarations of value identifiers and their signatures. For example, the *value* part of the SPACE scheme declares the function value ‘distance’ with the signature : \text{Point} \times \text{Point} \rightarrow \text{Real}. Here, except the signature, nothing about the function distance is specified in the *value* part. Its property is declared as an axiom in the *axiom* part.

Value definition can also be specified in the *value* part. For example, a function value \text{openBall} which returns an open sphere of radius \( r \) around a point \( p \) can be defined as shown in Fig. 4. (The precondition clause started with \texttt{pre} will be explained in Sect. 4.5.)

\[
\text{openBall}(r, p): \textbf{Real} \times \textbf{Point} \leadsto \textbf{Space} \\
\text{openBall}(r, p) \equiv \{ p' | p' : \textbf{Point} \land \text{distance}(p, p') < r \} \\
\texttt{pre} r > 0.0
\]

**Fig. 4.** RSL value definition of a function

RSL values correspond to HOL constants. Unlike in RAISE, we do not need to declare a constant with its signature separated from its definition. In HOL, Constants are defined using one of the following functions: \texttt{new_definition}, \texttt{new_specification}, \texttt{new_recursive_definition}. Figure 5 lists the HOL definition of the function \text{openBall}.

```haskell
res_quanLib.new_req_definition("OPEN_BALL_DEF", 
  (\texttt{pre} p: \textbf{Point}. \texttt{tr}: (\lambda r. r +\& 0), 
   (\texttt{OPEN_BALL} \ x \ p): "Space = \{ \ p' \mid \text{DISTANCE}(p, p') \land r \texttt{'}-\})
);
```

**Fig. 5.** HOL definition of \text{openBall}
4.4 Axiom Declarations

The axiom part of a module contains statements of properties of entities declared elsewhere in the module, e.g., in the value declarations. The axiom in the SPACE scheme shown in Fig. 2 states that the value returned by the distance function is non-negative, the distance function is symmetric and the distance between any two points \( p \) and \( p' \) is less than or equal to the sum of distances between these points and a third point \( p'' \). This asserts the property of the distance function. As mentioned in the previous section, this axiom and the value declaration together specify the value. This can be translated into HOL as constant definition or constant specification.

The axiom declarations and value definitions in RSL can be classified into two kinds: explicit and implicit. The explicit axioms and definitions in RSL correspond to constant definition in HOL. The definition of function ‘openBall’ shown in Fig. 4 is an explicit definition. An implicit axiom or definition only asserts certain properties of a function, for example the axiom of the function ‘distance’ shown in Fig. 2. The correspondence in HOL can be a constant specification. However, in order to define a constant using constant specification in HOL, a theorem asserting that the existence of a function having the required property has to be derived. This derivation generally cannot be automated. This posts a problem in automated translation from RSL to HOL.

4.5 Pre- and Post-expressions

The definition of the function ‘openBall’ also introduces the pre-condition expression which follows the keyword pre. This actually defines a partial function. The pre-condition restricts the domain of the function to all positive real numbers. We can map pre-conditions to restricted quantifications in HOL as already seen in Fig. 5.

RSL also allows functions to be defined abstractly as post-expressions. A post-expression asserts the properties of the returned value. For example, the function ‘separation’ is defined in Fig. 6 using a post-expression which states that the value \( sep \) returned by the function is the distance between two closest points in two spaces.

\[
\text{separation} : \text{Space} \times \text{Space} \rightarrow \text{Real} \\
\text{separation} (s, s') \text{ as } sep \\
\text{post} \\
\exists p_1, p_2 : \text{Point } \bullet p_1 \in s \land p_2 \in s' \land sep = \text{distance}(p_1, p_2) \Rightarrow \\
\forall p'_1, p'_2 : \text{Point } \bullet p'_1 \in s \land p'_2 \in s' \land sep \leq \text{distance}(p'_1, p'_2) \\
\text{pre } s \neq \text{emptySpace} \land s' \neq \text{emptySpace}
\]

**Fig. 6.** The definition of the function separation in RSL showing the use of post-expression
The corresponding HOL term for this post-expression is the \( \epsilon \) (select) term. A \( \epsilon \) term represents an object having the properties specified in the body of the term. This matches well with the RAISE post-condition. The HOL translation of the definition of the function 'separation' is listed in Fig. 7.

\[
\text{res_quanlib.new_resq_definition("SEPERATE_DEF",}
\]
\[
\text{(-'\!sp::\{s. \neg(s = EMPTY SPACE)\}. \!sp':\{s. \neg(s = EMPTY SPACE)\}.

\text{SEPARATION sp (sp'::Space) =}
\]
\[
\text{(@sep,}
\]
\[
\text{?p1 p2. (p1 IN sp) } \land \ (p2 IN sp') } \land
\]
\[
\text{(sep = DISTANCE(p1, p2)) -->}
\]
\[
\text{!p1' p2'. (p1' IN sp) } \land \ (p2' IN sp') } \land
\]
\[
\text{(sep \leq\! DISTANCE(p1', p2'))''--));}
\]

Fig. 7. The HOL translation of the function separation

5 The Proof Systems

The proof system of RSL consists of a large set of proof rules. Appendix B of [5] lists all 299 basic proof rules. This is because RSL is very large. In sharp contrast, HOL logic has only five axioms and eight primitive rules. All theorems can be derived from the axioms using these primitive inference rules.

However, the methods of deriving proofs in RAISE are similar to HOL. These are forward proofs and goal-directed proofs. In RAISE’s terminology, a goal is a justification condition or simply a condition. A proof is a justification, i.e., it justifies the truth of the condition. RAISE has two kinds of conditions:

- formal conditions are predicates whose truth have a formal significance;
- confidence conditions are predicates whose truth increases the confidence that there are no extreme cases, such as, array index out of bound, and so on.

The RAISE tool set contains a justification editor which is like a prover. The users can apply proof rules to break the condition down to simpler sub-conditions. If only proof rules are used to justify the conditions, a justification is formal. RAISE also allows an informal argument to be used to justify a condition. When one is convinced that 1) certain condition can be proved or, 2) it can be replaced by another condition, an informal argument can be used. An informal argument is, for the first case, an explanation giving the reason of the truth of the condition, and for the second case, an explanation followed by a new condition. This provides an escape route in the case the application is not too critical to insist on formally proving certain conditions. This is an example of the pragmatic approach that RAISE takes.
In HOL, all theorems are formally derived, and furthermore, the system actually performs all the inference steps in the derivation. Therefore, HOL is very formal and very reliable, i.e., theorems are trusted, but the development is very expensive. Although, it is possible to introduce axioms into HOL, but the users very rarely use this technique.

The granularity of a HOL proof is comparable to that of the RAISE justification editor. The user is required to guide the system to apply the appropriate tactics to the goal. With more than ten years collective effort of the HOL community, the latest version of the HOL system has a large number of proof tools to assist the users. Some of the most useful tools are some arithmetic decision procedures, first order logic reasoning tactics and a powerful simplifier. These speed up the proofs considerably.

6 Linking HOL and RAISE

To link HOL and RAISE, we need to consider three issues:

- the language issue,
- the communication issue,
- the security issue.

6.1 The Language Issue

Although both HOL and RSL are formal languages, they have different syntax and semantics. Furthermore, their proof systems are different. However, they have certain similarity. We have already described in the above sections the correspondence between RSL and HOL in an informal manner. The correspondence of the formal semantics of the two languages will need to be studied.

6.2 The Communication Issue

Both the HOL theorem prover and the RAISE tool set are interactive systems. In order to link the two systems, a communication interface has to be developed so that the specifications and proofs can be passed between them. However, because both system can operate in a batch mode to process information stored in files, therefore, at the moment we will not consider to develop a real-time interface between these systems.

6.3 The Security Issue

This issue deals with how the results from a system can be safely incorporated into another system. Since HOL theorems have been formally derived, they should be trustworthy to be incorporated into RAISE as a formal argument. Gunter [6] has suggested a method to incorporate theorems derived in other systems into HOL. This method adds a new primitive inference rule which
introduces theorems with a tag from trusted external systems into HOL. This method can be adopted by RAISE to incorporate HOL theorems as its justifications. This requires making some modification of the RAISE justification editor which cannot be done by a user.

Another way to incorporate theorems into RAISE is using informal arguments that is a mechanism in RAISE to introduce justification. As already mentioned in Sect. 5, informal argument may contain any description and a new condition. If undisciplined use of informal argument is allowed, the confidence in the correctness of the system will be reduced. If HOL theorems are incorporated into RAISE as informal arguments, the confidence on the correctness should not be reduced because HOL theorems are formally derived. As an alternative to modifying the RAISE justification editor, this method allows us to incorporate HOL theorems into RAISE.

7 Discussion and Conclusion

A pragmatic approach for incorporating HOL and RAISE is discussed in this paper. The incorporation of RAISE and HOL allows the two formal methods compensate and enhance with each other, and facilitate software engineering practitioners applying either of the methods whenever they feel that it is most appropriate or convenient. This is closer to the real life practice of engineers.

This incorporation approach, although does not bridge the two formal methods by their formal semantic foundations, provides a pragmatic mechanism for software engineers to use both methods in the same project. The coupling of the HOL modules and the corresponding RSL modules is low. Hence the work done in either of the methods will not affect one another. This follows the separate development principal of RAISE. The correctness of the bridging is managed by the stepwise development of RAISE.

HOL has been applied in the development of some non-software systems such as circuit design. Many systems consists of subsystems implemented in different technologies, e.g., some sub-systems are implemented in hardware while others are implement in software. The integration of different specification and verification environment is particular useful in the development of such systems.

Both RAISE and HOL communities are developing models for different application domains. By these inter-RAISE-HOL translations, the domain models can be shared by the two methods. This facilitates the re-use of developed domain models across different software engineering methods.

HOL has a set of more elegant inference rules. The overhead of proving is smaller than RAISE. By 'contracting out' RSL modules to HOL, the modules can be formally proved more conveniently. Hence, this can reduce the amount of informal justifications in RAISE. The reliability of the systems can be improved.

This incorporation of the two methods can also help software engineering practitioners, by taking advantage of HOL, to verify the correctness of their design. Furthermore, by taking the advantages of RAISE, software engineering
practitioners can obtain better project management on developing quality and reliable software systems.

There are still a number of issues to be studies. The first is the correspondence between the formal semantics of HOL and RSL. In order to preserve the meaning of the modules when they are translated between the two languages, it is very important to establish the formal semantic correspondence. The second is that there are difficulties in translating RSL implicit definitions into HOL automatically.

References

Effective Support for Mutually Recursive Types

Peter V. Homeier

Computer and Information Science Department, University of Pennsylvania
Philadelphia, Pennsylvania 19104-6389, USA
homeier@saul.cis.upenn.edu
http://www.cis.upenn.edu/~homeier

Abstract. For purposes of formal analysis, it is common to form a model of a system within a logic. This sometimes requires the introduction of new types which are mutually recursive. HOL90 has possessed for several years now two excellent libraries for mutually recursive types. Despite their powerful functionality, they are discovered to be difficult to use in practice. The input specifications of the mutually recursive types are laborious, the support for defining functions on these types is limited, and there is no built-in automated support for proving theorems about these types and functions, beyond proving the induction theorem. We address these software engineering issues in this paper, by the presentation of a new library, mutual, which includes all the definitional power of the others with a succinct interface and tools to facilitate the practical creation of function definitions and proofs. Researchers can now find this HOL90 software available from the Web.

1 Introduction

Modeling systems in HOL for study of their properties often requires the creation of new types in the logic. One of HOL’s strengths has been its powerful yet completely definitional and sound tools for creating and using new types, notably the excellent type definition package by Tom Melham [1]. This package provides facilities for specifying new recursive types in a concise syntax, automatically constructs the definitions required, and proves various theorems needed for using the new types, such as the type axiom, the structural induction theorem, the one-to-one and distinctiveness properties of the constructors, and the cases theorem. In addition to these theorems, the package also provides a tool for defining new functions on the new types, and a tactic for proving theorems about the new functions and types. This package has the appealing and enduring advantages of being easy to use, efficiently implemented, and completely sound.

In fact, if one were to look for a flaw in this package, the only place where one might reasonably criticize it might be in its scope. The package can only create one new recursive type at a time. This is fine for many applications, but there is a significant class of systems which evidence several types, where each type is defined in terms of itself and the others. These are called mutually recursive types. An example is the syntax structures of a programming language, where the syntax often is mutually recursive in interesting ways.
There are programming techniques that can be used to define these mutually recursive types using the standard type definition package. One new type is defined, which is a disjoint sum of all the mutually recursive types, with a tag to discriminate between the types. But these methods can be awkward to use, and do not provide the simplicity and ease-of-use that many users are familiar with from the standard package.

In 1991 Myra VanInwegen was working on her Ph.D. thesis [2] with Elsa Gunter, creating a definition of the syntax and semantics of SML within the HOL logic. SML is a language with mutually recursive syntax. To aid in representing this syntax by definitions of mutually recursive types, Gunter and VanInwegen created the mutrec library in the summer of 1991 [3]. This library was a significant addition to the functionality of HOL90, and provided impetus for users to switch to HOL90. Nevertheless, Gunter saw the need for additional functionality, and in the summer of 1992, Gunter jointly with Healfiene Goguen followed this library with the nestedrec library, with the ability to handle more general specifications of new types, including the use of pre-existing type operators such as list, prod, and sum in the specifications.

These new libraries provided new functionality that was greatly needed by many users of HOL who did not have the expertise to use the programming techniques mentioned before. However, these libraries came in a relatively rough condition, compared with the standard type definition package. Despite their useful functionality, these libraries were hard to use in practice, requiring laborious specifications of the types. In addition, the tool provided for creating definitions of new functions on the new types was restricted. With the most frequent impact, there was no tool provided analogous to the standard type definition package's \texttt{INDUCT_THEN} tactic, which helped to automate proofs of properties concerning a new type. One needed to use the induction theorem directly and manually, with a reduction in both ease and clarity.

In this paper we describe a new library for HOL, called mutual, which builds upon the functionality provided by the nestedrec library, providing tools to ease the creation and use of mutually recursive types, including nested recursion. The problems mentioned above are addressed, among other issues. This library makes direct use of the nestedrec library for creating the definitions, but adds functions to provide a more convenient and practical interface.

This new library adds no significantly new definitional functionality. Nevertheless, it can be considered a strict improvement over the pre-existing libraries. The thesis of this paper is that "ease-of-use" is an important feature of any package, which may be overlooked in the drive for increased functionality. The mutual library may be considered an illustrative example of this thesis.

The organization of this paper is as follows. Section 2 discusses previous approaches. In Section 3 we describe how to load the mutual library. Section 4 demonstrates the facilities for creating new definitions of mutually recursive types, including nested recursion. Section 5 describes the tool for defining new mutually recursive functions on these new types. Section 6 describes a tactic for proofs by mutual structural induction, and in Section 7 we conclude.
2 Previous Work

The fundamental tool for defining new types in HOL is \texttt{new_type_definition}, an ML function. This function requires the user to supply a theorem of the existence of values of the new type, and in addition create a bijection and its inverse between the new type and its representation. This involves a good deal of low-level detailed work that could be characterized as remote from the user’s intuitive conception of the type.

Probably the most commonly-used mechanism for defining new recursive types in HOL is the recursive type definition package by Tom Melham, as described in Chapter 20 of [1]. This package provides ML functions to define a single new concrete recursive type, with its constructor functions. The package also provides tools to produce theorems that state the axiomatization of the type, its induction principle, the disjointness and one-to-one principles of its constructors, and the cases theorem. New recursive functions in the HOL logic can be defined on the structure of this new type. In addition, the package provides the \texttt{INDUCT_THEN} tactic for proving properties about the new type and functions by structural induction.

Say we wished to define binary trees as either leaves or nodes with two child trees. A typical type definition in HOL88 would be

\begin{verbatim}
#let btreen_Axiom =
#  define_type
#    'btreen_Axiom' 'btreen = LEAF * | NODE btreen btreen';;
btreen_Axiom =
  |- ![f0 f1. 
      ![fn.
        (\x. fn(LEAF x) = f0 x) /
        (\b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)

The same type definition in HOL90 would be

- val btreen_Axiom =
  - define_type{ 
    name = "btreen_Axiom",
    type_spec = 'btreen = LEAF of 'a | NODE of btreen => btreen',
    fixities = [Prefix,Prefix] ];
val btreen_Axiom =
  |- ![f0 f1. 
      ![fn.
        (\x. fn(LEAF x) = f0 x) /
        (\b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)

This package has enjoyed great popularity, in no small part due to the excellent quality of the user interface provided and the efficient implementation of the tools. Last but not least, the documentation is complete and quite clear. Its obvious value has mandated its inclusion in the core HOL system, rather than as a library, to be readily available to all users.
This excellent package has only one significant limitation; it does not directly support mutually recursive types. To address this need, the \texttt{mutrec} library was created for HOL by Myra VanInwegen and Elsa Gunter in 1991. It provides a means to define mutually recursive types.

This brought the creation of mutually recursive types within the reach of many HOL users. However, Elsa Gunter was not satisfied with the functionality of this library, and working jointly with Healfdene Goguen, followed it a year later with an even more powerful library, \texttt{nestedrec}, which added the ability to refer to the new types being defined within some type operators, such as \texttt{list}, \texttt{sum}, and \texttt{prod}, so long as the proper theorems describing their axiomatization were also supplied.

Both these libraries, \texttt{mutrec} and \texttt{nestedrec}, were powerful additions to the set of tools in HOL for modeling general systems within the logic. However, these libraries also had certain weaknesses as well, in that they were not as well polished and easy to use as the standard recursive type definition package.

The most important areas needing improvement are these:

1. The specification of the input grammar is verbose, hard to compose and read, easy to get wrong, and very different from the simple input that the standard recursive type definition package requires.
2. When defining new functions on the new types, the functions are limited to exactly one argument, which must be one of the types defined.
3. No tactics are provided to aid in proofs by induction on the structure of the mutually recursive types, beyond proving the induction theorem.

Of these three, the first is the most obvious need; yet the last may be the most important, because for every new type definition, there may be many new functions defined, and for each new function defined, there may be many new properties proved about it.

3 Loading the Library

The \texttt{mutual} library is designed to reside in the \texttt{contrib} directory. Once installed, we load the \texttt{mutual} library by

\texttt{load_library_in_place (find_library "mutual");}

This will load several other libraries as well, including \texttt{mutrec} and \texttt{nestedrec}. Loading the \texttt{mutual} library will create the functors

\texttt{DefineMutualTypesFunc} and \texttt{StringDefineMutualTypesFunc},

and also the structure \texttt{mutualLib}. The functors are used to create new mutually recursive types; they vary only in whether they take a \texttt{term frag list} or a \texttt{string} as the input specification. The structure \texttt{mutualLib} has the signature
structure mutualLib :
  sig
    val define Mutual_functions
      : {def:term, fixities:fixity list option,
         name: string, rec axiom: thm}
      -> thm
    val MUTUAL_INDUCT THEN : thm -> thm tactic -> tactic
    val list_Axiom : thm
    val prod_Axiom : thm
    val sum_Axiom : thm
  end

This includes a function to define functions on the mutual types, a tactic to per-
form mutual structural induction, and three useful theorems for defining nested
mutually recursive types. Opening this structure makes these values available at
the top level:

- open mutualLib;
open mutualLib
val define Mutual_functions = fn
  : {def:term, fixities:fixity list option,
     name: string, rec axiom: thm} -> thm
val MUTUAL_INDUCT THEN = fn : thm -> thm tactic -> tactic
val list_Axiom =
  |- !x f. ?!fn. (fn1 [] = x) /
      (!h t. fn1 (CONS h t) = f (fn1 t) h t) : thm
val prod_Axiom = |- !f. ?!g. !x y. g (x,y) = f x y : thm
val sum_Axiom = |- !f g. ?!h. (!x. h (INL x) = f x) /
                   (!x. h (INR x) = g x) : thm

4 Definitions of Mutually Recursive Types

Mutually recursive types, with possible nesting of the recursion, are defined
using either the DefineMutualTypesFunc or StringDefineMutualTypesFunc
functors. This is best exhibited through an example. Consider the following
BNF grammar:

\[
\begin{align*}
  atexp & = \text{var} | \text{let dec in exp} \\
  exp & = atexp | exp atexp | match \\
  match & = \text{rule list} \\
  rule & = \text{pat} => \text{exp} \\
  dec & = \text{valbind} | \text{local dec in dec} | \text{dec ; dec} \\
  valbind & = \text{bind (pat to exp) list} | \text{rec valbind} \\
  \text{pat} & = \text{wild pat} | \text{var}
\end{align*}
\]

Figure 1 shows the need for mutual recursion by the presence of cycles.

If we represent the types of variables as a type variable 'var, then these
types may be defined as follows.
structure GramDef =
  DefineMutualTypesFunc
  (val name = "syntax"
   val recursor_thms = [list_Axiom, prod_Axiom]
   val types_spec =
     ' atexp = var_exp of 'var
        | let_exp of dec => exp ;

    exp = aexp of atexp
        | app_exp of exp => atexp
        | fn_exp of match ;

    match = match of rule list ;

    rule = rule of pat => exp ;

    dec = val_dec of valbind
        | local_dec of dec => dec
        | seq_dec of dec => dec ;

    valbind = bind of (pat # exp) list
              | rec_bind of valbind ;

    pat = wild_pat
        | var_pat of 'var ' ) ;

This closely matches the BNF presented above, and is an improvement over
the style of specifying such mutually recursive types in the nested_rec library.
Using that library requires one to create a structure with specific fields, including
a type specification with a recursive record structure. This is illustrated on the
next page, where the specification of the above example is given.
val var_ty = ("var" =>);

local
  structure Ast : NestedRecTypeInputSig =
  struct
    structure DefTypeInfo = DefTypeInfo
    open DefTypeInfo
    val def_type_spec =
      [{type_name = "atexp",
        constructors =
          [{name = "var_exp",
            arg_info = [existing var_ty],
            name = "let_exp",
            arg_info = [being_defined "dec",
              being_defined "exp"]}],
        type_name = "exp",
        constructors =
          [{name = "aexp",
            arg_info = [being_defined "atexp"]},
          {name = "app_exp",
            arg_info = [being_defined "exp",
              being_defined "atexp"]},
          {name = "fn_exp",
            arg_info = [being_defined "match"]}],
        type_name = "match",
        constructors =
          [{name = "match",
            arg_info = [type_op{Tyop="list",
              Args=[being_defined "rule"]}],
          type_name = "rule",
          constructors =
            [{name = "rule",
              arg_info = [being_defined "pat",
                being_defined "exp"]}],
          type_name = "dec",
          constructors =
            [{name = "val_dec",
              arg_info = [being_defined "valbind"]},
            {name = "local_dec",
              arg_info = [being_defined "dec",
                being_defined "dec"]},
            {name = "seq_dec",
              arg_info = [being_defined "dec",
                being_defined "dec"]}],
      type_name = "decl",
      constructors =
        [{name = "decl",
          arg_info = [being_defined "decl"]}]}]
{type_name = "valbind",
     constructors =
         [{name = "bind",
           arg_info = [type_op
                    {Tyop="list",
                     Args=[type_op
                          {Tyop="prod",
                           Args=[being_defined "pat",
                                 being_defined "exp"]}]]}],
           {name = "rec_bind",
            arg_info = [being_defined "valbind"]}]],
     type_name = "pat",
     constructors =
         [{name = "wild_pat",
           arg_info = []},
          {name = "var_pat",
           arg_info = [existing var_ty]]});

val recursor_thms = [list_Axiom, prod_Axiom]
val New_Ty_Existence_Thm_Name = "syntax_existence_thm"
val New_Ty_Induct_Thm_Name = "syntax_induction_thm"
val New_Ty_Uniqueness_Thm_Name = "syntax_uniqueness_thm"
val Constructors_Distinct_Thm_Name = "syntax_constructors_distinct"
val Constructors_One_One_Thm_Name = "syntax_constructors_one_one"
val Cases_Thm_Name = "syntax_cases"

end (* struct *)
in
 (* Prove the defining theorems for the type *)
 structure GramDef = NestedRecTypeFunc (Ast);
end;

The mutual library can condense the above specification due to the introduction of a parser for a mutually recursive types specification language. The language is modeled on that used in the standard HOL type definition package, and is the same except for having multiple type specifications, separated by semicolons. This parser is in fact very similar to the normal HOL90 parser, and could be integrated with it. The parser takes the specification as given in the shorter version above and parses it, creating the longer version seen above, which is then used as an argument in calling the nested_rec package.

The mutual library does give up some freedom present in nested_rec, for choosing the names of the theorems produced. In nested_rec, the six theorems are stored in the current theory under names which are specified independently
for each theorem. In the mutual library tools, only the root is specified by the user (in the above example, as the string "syntax") and the name of each theorem is created in a standard fashion by appending a standard suffix for that theorem, namely "exists," "induct," "unique," "distinct," "one_one," or "cases." This was chosen to ease the use of this tool and improve standardization of naming.

Note that the recursor theorems included with the specification must include the axiomatization theorems for all type operators used to nest types being defined, including new, user-defined type operators as well. It is a common error to leave some out; yet unnecessary ones may confuse the tool.

The DefineMutualTypesFunctor functor creates a new structure, as well as storing the six resulting theorems in the current theory. The new structure has signature DefTypeSig, and contains these theorems as well.

signature DefTypeSig =
  sig
    type thm
    val New_Ty_Induct_Thm : thm
    val New_Ty_Uniqueness_Thm : thm
    val New_Ty_Existence_Thm : thm
    val Constructors_Distinct_Thm : thm
    val Constructors_One_One_Thm : thm
    val Cases_Thm : thm
  end;

The actual theorems produced by the mutual library are not precisely the same as those produced by nested_rec. Some of the variable names generated automatically by the nested_rec tools were meaningless and hard to work with. Some we retained, like the long names for case functions, but for others, we generated more meaningful names based on the types of the variables, as in the standard recursive types package. In addition, the theorems were restructured and prepared for use by the other facilities of the mutual library. For the above example, the existence theorem generated by the mutual library is:

val New_Ty_Existence_Thm =
  |- !var_exp_case 1::case 1::case 1::case local_dec_case
      seq_dec_case aexp_case app_exp_case fn_exp_case
      match_case wild_pat_case var_pat_case
      atexp_dec_exp_match_pat_rule_valbind_ch44_pat_exp_case
      atexp_dec_exp_match_pat_rule_valbind_NIL_pat_exp_prod
      atexp_dec_exp_match_pat_rule_valbind_NIL_rule_case
      atexp_dec_exp_match_pat_rule_valbind_CONS_pat_exp_prod
      atexp_dec_exp_match_pat_rule_valbind_CONS_rule_case
      rule_case
      atexp_dec_exp_match_pat_rule_valbind_NIL_rule_case
      atexp_dec_exp_match_pat_rule_valbind_CONS_rule_case
      bind_case rec_bind_case.
?fn id fn e fm fn0 fnp fn1 fnl0 fnr fnl1 fnv.
  (!x. fn a (var_exp x) = var_exp_case x) /
  (!d e. fn a (let_exp d e) =
    let_exp_case (fnid d) (fne e) d e) /
  (!v. fn d (val_dec v) = val_dec_case (fnnv v) v) /
  (!d0 d1. fn (local_dec d0 d1) =
    local_dec_case (fnid d0) (fnd d1) d0 d1) /
  (!d0 d1. fn (seq_dec d0 d1) =
    seq_dec_case (fnid d0) (fnd d1) d0 d1) /
  (!a. fn e (aexp a) = aexp_case (fna a) a) /
  (!e a. fn e (app_exp e a) =
    app_exp_case (fne e) (fna a) e a) /
  (!m. fn (fn_exp m) = fn_exp_case (fnfm m) m) /
  (!l. fnm (match l) = match_case (fnl1 l) l) /
  (fn0 wild_pat = wild_pat_case) /
  (!x. fnp0 (var_pat x) = var_pat_case x) /
  (!p e.
    fnp1 (p, e) =
    atexp_dec_exp_match_pat_rule_valbind_ch44_pat_exp_case
    (fnp0 p) (fne e) p e) /
  (fnl0 []) =
    atexp_dec_exp_match_pat_rule_valbindNIL_pat_exp_prod
    atexp_dec_exp_match_pat_rule_valbind_case) /
  (!p l.
    fnl0 (CONS p l) =
    atexp_dec_exp_match_pat_rule_valbindCONS_pat_exp_prod
    atexp_dec_exp_match_pat_rule_valbind_case
    (fnp1 p) (fnl1 1) l 1) /
  (!p e. fnr (rule p e) =
    rule_case (fnp0 p) (fne e) p e) /
  (fnl1 []) =
    atexp_dec_exp_match_pat_rule_valbindNIL_rule_case) /
  (!r l.
    fnl1 (CONS r l) =
    atexp_dec_exp_match_pat_rule_valbindCONS_rule_case
    (fnr r) (fnl1 1) r l) /
  (!l. fnv (bind l) = bind_case (fnl0 l) l) /
  (!v. fnv (rec_bind v) = rec_bind_case (fnv v) v) : thm

Where the above existence theorem has
?fn id fn e fm fn0 fnp fn1 fnl0 fnr fnl1 fnv.
the corresponding theorem generated by the nested_rec library has instead
?y y' y'' y''' y'''' y''''' y'''''' y''''''' y'''''''' y''''''''' y'''''''''' y''''''''''' y'''''''''''' y'''''''''''''
with corresponding substitutions throughout.
5 Defining Mutually Recursive Functions

Once the mutually recursive types are defined, we can now define a cooperating set of mutually recursive functions on them. define_mutual_functions is used for this, as in the following example. This example defines functions to return the variables in a phrase of the language, except for those in a given set \( s \).

```plaintext
val vars_thm = define_mutual_functions
{name = "vars_thm",
 rec_axiom = syntax_exists,
 fixities = NONE,
 def =
  (\--'(atexpV (var_exp (v:'var)) s = (v IN s => {} | \{v\})) \/
    (atexpV (let_exp d e) s = (decV d s) UNION (expV e s)) \/
    (expV (aexp a) s = atexpV a s) \/
    (expV (app_exp e a) s = (expV e s) UNION (atexpV a s)) \/
    (expV (fn_exp m) s = matchV m s) \/
    (matchV (match rs) s = matchVs rs s) \/
    (matchVs (NIL) s = \{\}) \/
    (matchVs (CONS r mrst) s = (ruleV r s) UNION (matchVs mrst s)) \/
    (ruleV (rule p e) s = (patV p s) UNION (expV e s)) \/
    (decV (val_dec b) s = valbindV b s) \/
    (decV (local_dec d1 d2) s = (decV d1 s) UNION (decV d2 s)) \/
    (decV (seq_dec d1 d2) s = (decV d1 s) UNION (decV d2 s)) \/
    (valbindV (bind bs) s = valbindVs bs s) \/
    (valbindV (rec_bind vb) s = (valbindV vb s)) \/
    (valbindVs NIL s = \{\}) \/
    (valbindVs (CONS hhd brst) s = (valbindVp hhd s) UNION (valbindVs brst s)) \/
    (valbindVp (p,e) s = (patV p s) UNION (expV e s)) \/
    (patV wild_pat s = \{\}) \/
    (patV (var_pat v) s = (v IN s => {} | \{v\})'--));
```
This creates the following definition:

```ml
val vars_thm =
  |- (!v s. atexpV (var_exp v) s = ((v IN s) => {} | {v})) /
    (!d e s. atexpV (let_exp d e) s = decV d s UNION expV e s) /
    (!a s. expV (aexp a) s = atexpV a s) /
    (!e s. expV (app_exp e a) s = expV e s UNION atexpV a s) /
    (!m s. expV (fn_exp m) s = matchV m s) /
    (!rs s. matchV (match rs) s = matchVs rs s) /
    (!s. matchVs [] s = {}) /
    (!r mrst s. matchVs (CONS r mrst) s =
      ruleV r s UNION matchVs mrst s) /
    (!p e s. ruleV (rule p e) s = patV p s UNION expV e s) /
    (!b s. decV (val_dec b) s = valbindV b s) /
    (!d1 d2 s. decV (local_dec d1 d2) s =
      decV d1 s UNION decV d2 s) /
    (!d1 d2 s. decV (seq_dec d1 d2) s =
      decV d1 s UNION decV d2 s) /
    (!bs s. valbindV (bind bs) s = valbindVs bs s) /
    (!vb s. valbindV (rec_bind vb) s = valbindV vb s) /
    (!s. valbindVs [] s = {}) /
    (!bhd brst s. valbindVs (CONS bhd brst) s =
      valbindVp bhd s UNION valbindVs brst s) /
    (!p e s. valbindVp (p,e) s = patV p s UNION expV e s) /
    (!s. patV wild_pat s = {}) /
    (!v s. patV (var_pat v) s = ((v IN s) => {} | {v})): thm
```

This theorem matches the specification, including the names of the variables used. This is not the case for the nested_rec library. Also note the additional argument `s` to each function. Any number of arguments may be added, but the first argument must be one of the recursive types. It is possible to define functions on only one or some of the types defined in a mutual set; not all need be present in the function definition. However, note that if any of the constructors of a type are present, they must all be present, unless the last pattern for the type is the variable “allelse”.

The nested_rec version of define_mutual_functions supports only one argument. Nevertheless, we can still define the same functions by moving the extra arguments to be lambda abstractions on the right hand side. However, the resulting theorem is different in its structure and names used, as illustrated below:

```ml
val vars_thm =
  |- (!x1. atexpV (var_exp x1) = vars (x1 IN s) => {} | {x1}) /
    (!x1 x2. atexpV (let_exp x1 x2) =
      vars (decV x1 s UNION expV x2 s) /
    (!x1. expV (aexp x1) = vars (atexpV x1 s) /
    (!x1 x2. expV (app_exp x1 x2) =
      vars (expV x1 s UNION atexpV x2 s)) /
```
\begin{verbatim}
(!x1. expV (fn_exp x1) = (\s. matchV x1 s)) /
(!x1. matchV (match x1) = (\s. matchVs x1 s)) /
(matchVs [] = (\s. ())) /
(!x1 x2. matchVs (CONS x1 x2) =
  (\s. ruleV x1 s UNION matchVs x2 s)) /
(!x1 x2. ruleV (rule x1 x2) =
  (\s. patV x1 s UNION expV x2 s)) /
(!x1. decV (val_dec x1) = (\s. valbindV x1 s)) /
(!x1 x2. decV (local_dec x1 x2) =
  (\s. decV x1 s UNION decV x2 s)) /
(!x1 x2. decV (seq_dec x1 x2) =
  (\s. decV x1 s UNION decV x2 s)) /
(!x1. valbindV (bind x1) = (\s. valbindVs x1 s)) /
(!x1. valbindV (rec_bind x1) = (\s. valbindV x1 s)) /
(valbindVs [] = (\s. ())) /
(!x1 x2. valbindVs (CONS x1 x2) =
  (\s. valbindVs x1 s UNION valbindVs x2 s)) /
(!x1 x2. valbindVp (x1,x2) =
  (\s. patV x1 s UNION expV x2 s)) /
(patV wild_pat = (\s. ())) /
(!x1. patV (var_pat x1) = (\s. (x1 IN s) => {} | {x1}))

: thm
\end{verbatim}

This structure obliges one to use beta reduction when using the definition theorem, rather than simple rewriting.

6 Proofs by Mutual Structural Induction

The third and final part of the mutual library is the support for proofs of mutual structural induction, through \texttt{MUTUAL\_INDUCT\_TAC}. This is a revised version of the \texttt{INDUCT\_TAC} written by Tom Melham in the standard recursive types package, expanded for mutually recursive types. There is much care taken in the original version to break the current goal into a practical and convenient set of subgoals according to the induction principle, and we have tried to preserve this quality.

The ML function \texttt{MUTUAL\_INDUCT\_TAC} has type

\[ \texttt{thm \rightarrow (thm \rightarrow tactic) \rightarrow tactic} \]

and can be used to generate a structural induction tactic for a set of concrete types definable using the functors of Section 4. The first argument is an induction theorem of the form created by these functors. The second argument is a theorem continuation that determines what is to be done with the induction hypotheses when the resulting tactic is applied to a goal.

If \texttt{ind\_th} is an induction theorem for a set of mutually recursive concrete types \(op_1, \ldots, op_n\), where this includes all auxiliary types arising through the nesting of types in the definition, and if each concrete type \(op_i\) has \(m_i\) constructors \(C_1^i, \ldots, C_{m_i}^i\), and \(F\) is a theorem continuation, then the tactic
**MUTUAL_INDUCT_THEN** *indₜ ℎ F*

will reduce a goal of the form

\[(Γ, (→' (∀x₁ : op₁. t₁[x₁]) ∧ \ldots (∀xₙ : opₙ. tₙ[xₙ]) \rightarrow ))\]

to a collection of (possibly) \(\sum_{i=1}^{n} m_i\) induction subgoals (this count may not be precise for various reasons). The goal may list the conjuncts in any order: they need not be in the precise same order as the corresponding clauses listed in the induction theorem *indₜ ℎ*. In fact, some of the goal clauses may be missing entirely, in which case the tactic will presume that they are \((∀x_i : op_i. T)\).

As an example, consider proving that for the variable-collecting functions defined earlier, none of them collect any variables in the exclusion set *s*.

\[
g '(!s (x:'var). x \text{ IN atexpV a s} \rightarrow '(x \text{ IN s})) /

(!s (x:'var). x \text{ IN expV e s} \rightarrow '(x \text{ IN s})) /

(!s (x:'var). x \text{ IN matchV m s} \rightarrow '(x \text{ IN s})) /

(!rs s (x:'var). x \text{ IN matchVs rs s} \rightarrow '(x \text{ IN s})) /

(!r s (x:'var). x \text{ IN ruleV r s} \rightarrow '(x \text{ IN s})) /

(!d s (x:'var). x \text{ IN decV d s} \rightarrow '(x \text{ IN s})) /

(!v s (x:'var). x \text{ IN valbindV v s} \rightarrow '(x \text{ IN s})) /

(!l s (x:'var). x \text{ IN valbindVs l s} \rightarrow '(x \text{ IN s})) /

(!pr s (x:'var). x \text{ IN valbindVp pr s} \rightarrow '(x \text{ IN s})) /

(!p s (x:'var). x \text{ IN patV p s} \rightarrow '(x \text{ IN s}));
\]

These clauses are listed in an order similar to the definition, which is convenient. These can be simultaneously broken into cases by mutual structural induction with the following tactic:

- `e(MUTUAL_INDUCT_THEN syntax_induct ASSUME_TAC);
```
OK.
19 subgoals:
val it =
```
\[
(\rightarrow'!s x. x \text{ IN valbindW (rec_bind v) s} \rightarrow '(x \text{ IN s})'),
```

```
(\rightarrow'!s x. x \text{ IN valbindW v s} \rightarrow '(x \text{ IN s}'))
```

```
(\rightarrow'!s x. x \text{ IN valbindW (bind l) s} \rightarrow '(x \text{ IN s}'))
```

```
(\rightarrow'!s x. x \text{ IN valbindWs l s} \rightarrow '(x \text{ IN s}'))
```

```
(\rightarrow'!s x. x \text{ IN matchVs (CONS r rs) s} \rightarrow '(x \text{ IN s}'))
```

```
(\rightarrow'!s x. x \text{ IN ruleW r s} \rightarrow '(x \text{ IN s}'))
```

```
(\rightarrow'!s x. x \text{ IN matchVs rs s} \rightarrow '(x \text{ IN s}'))
```

-
(-'! s x. x IN matchVs [] s => ~(x IN s)''--)

(-'! s x. x IN ruleV (rule p e) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN patV p s => ~(x IN s)''--)
   (-'! s x. x IN expV e s => ~(x IN s)''--)

(-'! s x. x IN valbindVs (CONS pr l) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN valbindVp pr s => ~(x IN s)''--)
   (-'! s x. x IN valbindVs l s => ~(x IN s)''--)

(-'! s x. x IN valbindVs [] s => ~(x IN s)''--)

(-'! s x. x IN valbindVp (p,e) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN patV p s => ~(x IN s)''--)
   (-'! s x. x IN expV e s => ~(x IN s)''--)

(-'! s x. x' IN patV (var_pat x) s => ~(x' IN s)''--)

(-'! s x. x IN patV wild_pat s => ~(x IN s)''--)

(-'! s x. x IN matchV (match rs) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN matchVs rs s => ~(x IN s)''--)

(-'! s x. x IN expV (fn_exp m) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN matchV m s => ~(x IN s)''--)

(-'! s x. x IN expV (app_exp e a) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN expV e s => ~(x IN s)''--)
   (-'! s x. x IN atexpV a s => ~(x IN s)''--)

(-'! s x. x IN expV (aexp a) s => ~(x IN s)''--)
-------------------------------------------------
   (-'! s x. x IN atexpV a s => ~(x IN s)''--)

Peter V. Homeier

(--' s x. x IN decV (seq_dec d d') s ==> ~(x IN s)'--)
-------------
  (--' s x. x IN decV d s ==> ~(x IN s)'--)
  (--' s x. x IN decV d' s ==> ~(x IN s)'--)

(--' s x. x IN decV (local_dec d d') s ==> ~(x IN s)'--)
-------------
  (--' s x. x IN decV d s ==> ~(x IN s)'--)
  (--' s x. x IN decV d' s ==> ~(x IN s)'--)

(--' s x. x IN decV (val_dec v) s ==> ~(x IN s)'--)
-------------
  (--' s x. x IN atexpV (let_exp d e) s ==> ~(x IN s)'--)

(--' s x. x IN atexpV (var_exp x) s ==> ~(x IN s)'--)

: goalstack

In fact, the original goal can be entirely proven by the tactic

e MUTUAL_INDUCT_THEN syntax_induct ASSUME_TAC
  THEN REWRITE_TAC[vars_thm]
  THEN REPEAT GEN_TAC
  THEN ((REWRITE_TAC[theorem "set" "IN_UNION"]
    THEN REWRITE_TAC[theorem "set" "NOT_IN_EMPTY"]
    THEN STRIP_TAC
    THEN RES_TAC
    THEN NO_TAC)
  ORELSE
  (COND_CASES_TAC
    THEN REWRITE_TAC[theorem "set" "IN_INSERT",
                    theorem "set" "NOT_IN_EMPTY"]
    THEN DISCH_TAC
    THEN ASM_REWRITE_TAC[])
);

In the nested_rec library there was no analogous tactic provided. The only thing we could find was an info@hol posting by Myra VanInwegen, dated March 19, 1996, where she wrote:
We didn't include such a tactic with the package, but obviously, one is needed to prove properties of mutually recursive types. This is what I use:

(* for now, the things proven must be in the same order as in the conclusion of the induction theorem *)
fun mutual_induct induct_thm (asms, gl) =
  let val props_list = map
      (fn tm => mk_abs (dest_forall tm)) (strip_conj gl)
  val speced_ind = BETA_RULE (SPECL props_list induct_thm)
  in
  MP_IMP_TAC speced_ind (asms, gl)
  end

The only problem with it is, as I note in the comment, that the properties have to be in the same order as those in the conclusion of the induction theorem. The result of applying this function is one subgoal that is a big conjunction, with each conjunct being a case in the induction.

Using the mutual_induct function, we can prove a similar result as before. The goal must be reordered, and the tactic must make use of BETA_TAC. The resulting tactic is slightly larger than the previous one. To compare these two tactics, where MUTUAL_INDUCT_THEN presents the user with

(\(\forall x.x\ IN\ atexpV\ (let\_exp\ d\ e)\ s\ \Rightarrow\ \sim(x\ IN\ s)\)\)

-----------------------------
(\(\forall x.x\ IN\ decV\ d\ s\ \Rightarrow\ \sim(x\ IN\ s)\)\)
(\(\forall x.x\ IN\ expV\ e\ s\ \Rightarrow\ \sim(x\ IN\ s)\)\)

mutual_induct followed by REPEAT CONJ_TAC presents

(\(\forall y.y\)\)

(\(\forall x.x\ IN\ decV\ y\ s\ \Rightarrow\ \sim(x\ IN\ s)\)\)
(\(\forall x.x\ IN\ expV\ y\ s\ \Rightarrow\ \sim(x\ IN\ s)\)\)
(\(\forall x.x\ IN\ atexpV\ (let\_exp\ y\ s\)\ s\ \Rightarrow\ \sim(x\ IN\ s)\)\)

These \(y\)\ variables appear to be an artifact of the implementation of the nested_rec library.

7 Summary and Conclusions

We have defined a new library within HOL, mutual, to support the creation and use of mutually recursive types with nesting. This is essentially equivalent to the functionality of the nested_rec library, but adds facilities to ease its use in practical ways.
The input specifications are shorter and clearer, close to the BNF form, and similar to the syntax required for the non-mutual recursive type definition package. Functions can be defined on these types with more arguments. Properties may be proved by mutual structural induction, supported by a general-purpose function for these tactics.

The mutual library software is currently available for HOL90 versions 7 and 10, through the Web page at


For all these tools, feedback is welcome and encouraged, as we would like to polish them for general use. Please notify the author if this library is adapted to another environment, so it can be posted here as well.

Caution: this software should be considered only of beta quality, and may contain errors. It is being released now in order to support researchers for whom this level of quality is acceptable, and who may be able to help in testing and improving this software.

This exercise is perhaps best appreciated as an investigation into the relative importance of ease-of-use. This is not a question with a precise answer, but depends on people's preferences. Thus this paper is only an entry in the ongoing discussion.

DEDICATION: This paper is dedicated to David F. Martin, Professor and Founding Member of Computer Science at UCLA, who passed away December 22, 1996. Without his encouragement and involvement, all of my future career would not be.

Soli Deo Gloria.

References


Author Index

Benzmüller, Christoph ............... 1
Black, Paul ....................... 19
Curzon, Paul ...................... 31
Dawson, Jeremy ................... 47
Gravell, Andrew .................... 63, 73
Gunter, Elsa ....................... 85
Homeier, Peter .................... 141
Leung, Karl ....................... 127
Marques, François .................. 95

Melham, Tom ..................... 105
Mohamed, Otmane Aït .......... 31
Pratten, Chris .................... 63, 73
Susanto, Kong Woei ............. 105
Tahar, Sofiène .................... 31
Watson, Geoffrey ............... 119
Windley, Phillip ................ 19
Wong, Wai ....................... 127