Lecture Notes in Computer Science 1690
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Yves Bertot   Gilles Dowek   André Hirschowitz
Christine Paulin   Laurent Théry (Eds.)

Theorem Proving in Higher Order Logics

12th International Conference, TPHOLs '99
Nice, France, September 14-17, 1999
Proceedings

Springer

Volume Editors

Yves Bertot
Gilles Dowek
Laurent Théry
INRIA Sophia Antipolis
2004 Route des Lucioles, F-06902 Sophia Antipolis Cedex, France

André Hirschowitz
University of Nice - Sophia Antipolis
Parc Valrose, F-06108 Nice Cedex 2, France

Christine Paulin
University of Paris XI
15, rue Georges Clemenceau, F-91405 Orsay Cedex, France

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Theorem proving in higher order logics : 12th international
conference ; proceedings / TPHOLs '99, Nice, France, September
14 - 17, 1999 / Yves Bertot . . . (ed.). - Berlin ; Heidelberg ; New York
; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ;
Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1690)
ISBN 3-540-66463-7

CR Subject Classification (1998): F.4.1, F.3.1, I.2.3, D.2.4, B.6.3

ISSN 0302-9743
ISBN 3-540-66463-7 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10704460 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

This book contains the proceedings of *the 12th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs'99), which was held in Nice at the University of Nice-Sophia Antipolis, September 14{17, 1999. Thirty-five papers were submitted as completed research, and each of them was refereed by at least three reviewers appointed by the program committee. Twenty papers were selected for publication in this volume.

Following a well-established tradition in this series of conferences, a number of researchers also came to discuss work in progress, using short talks and displays at a poster session. These papers are included in a supplementary proceedings volume. These supplementary proceedings take the form of a book published by INRIA in its series of research reports, under the following title : *Theorem Proving in Higher Order Logics: Emerging Trends 1999*.

The organizers were pleased that Dominique Bolignano, Arjeh Cohen, and Thomas Kropf accepted invitations to be guest speakers for TPHOLs'99. For several years, D. Bolignano has been the leader of the VIP team in the Dyade consortium between INRIA and Bull and is now at the head of a company *Trusted Logic*. His team has been concentrating on the use of formal methods for the effective verification of security properties for protocols used in electronic commerce. A. Cohen has had a key influence on the development of computer algebra in The Netherlands and his contribution has been of particular importance to researchers interested in combining the several known methods of using computers to perform mathematical investigations. T. Kropf is an important actor in the Europe-wide project PROSPER, which aims to deliver the benefits of mechanized formal analysis to system builders in industry. Altogether, these invited speakers gave us a panorama of applications for theorem proving and discussed its impact on the progress of scientific investigation as well as technological advances.

This year has confirmed the evolution of the conference from HOL-users' meeting to conference with a larger scope, spanning over uses of a variety of theorem proving systems, such as Coq, Isabelle, Lambda, Lego, NuPrl, or PVS, as can be seen from the fact that the organizers do not belong to the HOL-user community.

Since 1993, the proceedings have been published by Springer-Verlag as Volumes 780, 859, 971, 1125, 1275, 1479, and 1690 of *Lecture Notes in Computer Science*. The conference was sponsored by the laboratory of mathematics of the University of Nice-Sophia Antipolis, Intel, France Telecom, and INRIA.

September 1999

Yves Bertot, Gilles Dowek,
Andre Hirschowitz, Christine Paulin,
Laurent Thery

Organization

Yves Bertot (INRIA)
Gilles Dowek (INRIA)
Andre Hirschowitz (Universite de Nice)
Christine Paulin (Universite de Paris-Sud)
Laurent Thery (INRIA)

Program Committee

Mark Aagaard (Intel)
Sten Agerholm (IFAD)
David Basin (Freiburg)
Yves Bertot (INRIA)
Richard Boulton (Edinburgh)
Gilles Dowek (INRIA)
Mike Gordon (Cambridge)
Jim Grundy (ANU)
Elsa Gunter (Lucent)
Joshua Guttman (Mitre)
John Harrison (Intel)
Doug Howe (Lucent)
Bart Jacobs (Nijmegen)
Sara Kalvala (Warwick)
Tom Melham (Glasgow)
Paul Miner (NASA)
Malcolm Newey (ANU)
Topbias Nipkow (Munich)
Sam Owre (SRI)
Christine Paulin-Mohring (Paris, *Chair*)
Lawrence Paulson (Cambridge)
Sofiene Tahar (Concordia)

Invited Speakers

Dominique Bolignano (Trusted Logic)
Arjeh M. Cohen (T.U. Eindhoven)
Thomas Kropf (Tübingen)

Additional Reviewers

O. Ait-Mohamed
C. Ballarin
G. Bella
J. Courant
P. Curzon
D. Cyrluk
P. Dybjer
J.-C. Filliâtre
M.-S. Jahanpour
F. Kammüller
P. Lincoln
A. Mader
O. Müller
A. Pitts
E. Poll
H. Rue
K. Sunesen
D. Syme
H. Tews
J. Thayer
M. Wenzel

Table of Contents

Recent Advancements in Hardware Verification – How to Make Theorem Proving Fit for an Industrial Usage 1
Thomas Kropf

Disjoint Sums over Type Classes in HOL 5
Norbert Völker

Inductive Datatypes in HOL – Lessons Learned in Formal-Logic Engineering 19
Stefan Berghofer and Markus Wenzel

Isomorphisms – A Link Between the Shallow and the Deep 37
Thomas Santen

Polytypic Proof Construction 55
Holger Pfeifer and Harald Rueß

Recursive Function Definition over Coinductive Types 73
John Matthews

Hardware Verification Using Co-induction in COQ 91
Solange Coupet–Grimal and Line Jakubiec

Connecting Proof Checkers and Computer Algebra Using *OpenMath* 109
Olga Caprotti and Arjeh M. Cohen

A Machine-Checked Theory of Floating Point Arithmetic 113
John Harrison

Universal Algebra in Type Theory 131
Venanzio Capretta

Locales – A Sectioning Concept for Isabelle 149
Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson

Isar – A Generic Interpretative Approach to Readable Formal Proof Documents 167
Markus Wenzel

On the Implementation of an Extensible Declarative Proof Language 185
Vincent Zammit

Three Tactic Theorem Proving 203
Don Syme

Mechanized Operational Semantics via (Co)Induction 221
Simon J. Ambler and Roy L. Crole

Representing WP Semantics in Isabelle/ZF 239
Mark Staples

A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata 255
Klaus Schneider and Dirk W. Hoffmann

From I/O Automata to Timed I/O Automata 273
Bernd Grobauer and Olaf Müller

Formal Methods and Security Evaluation (*Invited Talk*) 291
Dominique Bolignano

Importing MDG Verification Results into HOL 293
Haiyan Xiong, Paul Curzon, and Sofiéne Tahar

Integrating Gandalf and HOL 311
Joe Hurd

Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving 323
Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger

Symbolic Functional Evaluation... 341
Nancy A. Day and Jeffrey J. Joyce

Author Index 359

Recent Advancements in Hardware Verification { How to Make Theorem Proving Fit for an Industrial Usage

Thomas Kropf

Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen, Sand 13, D-72076 Tübingen, Germany
kropf@informatik.uni-tuebingen.de

1 Motivation

One of the predominant applications of formal methods including theorem proving is the verification of both software and hardware. However, whereas software verification must still be considered mainly as an academic exercise, hardware verification has become an established technique in industry. Commercial hardware verification tools are available from leading EDA (electronic design automation) tool vendors and are an essential part of many established digital design flows. This contrasts to the fact that software verification has been an ongoing research topic for a considerably longer time compared to activities in hardware verification.

Where does this difference in acceptance come from? The key components for a new approach to be successful in practice are: a suitable theoretical foundation, the existence of tools, the acceptance of the target users, a smooth integration into existent work flows as well as the applicability to real-world examples, together leading to a measurable productivity increase. Undoubtedly, the simplicity of the theories necessary for many tasks in hardware verification, i.e., propositional logic and finite state machines, justify a good part of its success. The available decision procedures allow fully automated tools which can be easily used also by non-logicians.

However, looking at the commercially successful hardware verification tools like equivalence checkers or model checkers, it becomes visible that the other aspects also play a significant role. The tools are applicable to hardware designs, written in standardized hardware description languages like Verilog or VHDL. Hence, they can be easily integrated into existent design flows together with other validation tools like simulators. They are targeted at the verification of large circuits at the expense of proof power, e.g., often only a simple equivalence check of two combinational circuits can be performed but on very large circuits. Moreover, it has turned out that the usual application scenario is not the successful correctness proof of a finished hardware component. It is rather the case that almost all verification runs fail due to wrong specifications or due to design errors still present in the module. Thus, verification is mainly treated as an debugging aid. As a consequence, an integral part of a verification tool consists

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 1{4, 1999.

of means to trace down potential error sources. The usual approach allows the generation of counterexamples, i.e., input patterns or sequences which lead to an erroneous output behavior.

Although being very successful from a commercial point of view, the currently available hardware verification tools still have many deficiencies. Although applicable to circuits of considerable size, they still do not scale-up in a sufficient way, e.g., by a suitable use of divide-and-conquer strategies. As soon as they go beyond a simple equivalence check, formal specifications have to be given which in most cases exceed the capabilities of industrial hardware designers. Moreover, the tools are targeted only at specific verification tasks, e.g., the verification of controllers or the verification of arithmetic data paths. Thus, they are unable to verify circuits containing both kind of modules. The counterexamples produced in case of a failed verification run still require a tedious simulation-based search of the real fault cause.

2 Solutions

2.1 Key Components

The deficiencies described above require the creation of a new generation of verification environments. Whereas fully automated tools each rely on a certain decision procedure, a divide-and-conquer approach requires the use of inference rules, provided by theorem proving systems. In contrast to existent systems an industrially usable theorem proving approach must be applicable to large problems and must be usable without much knowledge of the underlying formal system. Hence, means to "hide" the theorem proving part from the user have to be found. Besides suitable user interfaces this also comprises new way to state formal specifications in an informal way. As stated above complex hardware designs may require the combined usage of different verification tools. This requires the engineering of proper interaction means both from a semantic and from a tool programming point of view. The last point concerns the finding of errors. The derivation of counterexamples, i.e., input stimuli, is only based on a functional view of the circuit under consideration and does not take into account the known topological structure of the hardware design to narrow down the error source to certain components.

2.2 The PROSPER Project

The PROSPER project (Proof and Specification Assisted Design Environments)[1], funded by the European Commission as part of the ESPRIT Framework IV program[2], addresses all of the issue described in the last section. It is aimed at the research and development of the technology needed to deliver the benefits of mechanized formal specification and verification to system designers

[1] http://www.dcs.gla.ac.uk/prosper/

[2] ESPRIT project LTR 26241

in industry. The project partners[3] will try to provide examples of the next generation of EDA and CASE tools, incorporating user-friendly access to formal techniques.

The key component of PROSPER is an open proof architecture, created around a theorem prover, called the core proof engine (CPE). This engine is based on higher-order logic and on many principles of the HOL theorem proving system. It is enriched by a standardized interface for other proof tools, which can be added as "plug-ins" to enhance the proof power and the degree of automation of the CPE. This also allows the combined use of different proof tools to tackle a certain verification task. The interface has been designed in such a way that already existent proof tools like model checkers or first-order theorem provers can be easily added to the overall system. It is currently based on a communication via sockets and on standardized abstract data types, the latter based on a HOL syntax of terms. The CPE and the proof tool plug-ins together may then be used as the formal methods component within a commercial EDA design tool or a CASE tool. The intended example applications are a hardware verification workbench and a VDM-SL tool box. To ease the use of formal methods for programmers and hardware designers with only limited knowledge of formal methods, another important part is the engineering of suitable interfaces. This comprises approaches to translate natural language descriptions of circuit specifications into temporal logic as well as new means in identifying faulty circuit components in case of a failed verification run.

2.3 The Hardware Verification Part of PROSPER

One exemplar application of the PROSPER concept is the creation of a hardware verification workbench. It is able to provide correctness proofs for hardware systems, described in the standardized hardware description language VHDL and Verilog. This application is ideally suited to demonstrate the interplay between the CPE and proof plug-ins like tautology checkers or model checkers, all needed to play together to verify complex digital circuits.

To be able to provide one uniform proof environment, the common semantic basis of both languages VHDL and Verilog is identified and formalized by an intermediate language IL. Thus, by first translating a given circuit description into IL, the same proof strategies may be used for both languages. The actual proof is usually performed with decision procedures, using the CPE mainly to decompose complex proof goals.

It has turned out that often a simple counterexample generation is insufficient to easily find faulty components. As part of the PROSPER project new techniques are established to analyze the structure of a given circuit with regard to a given functional specification. This allows to automatically identify and to correct faulty components in case the circuit does not behave according to the specification.

[3] The current project partners are the Universities of Glasgow, Cambridge, Edinburgh (all UK),Tbingen (Germany) as well as IFAD (Denmark) and Prover Technology (Sweden).

It is hoped that the PROSPER approach will lead theorem proving to a similar acceptance in industry as equivalence checking and model checking techniques have already today.

Disjoint Sums over Type Classes in HOL

Norbert Völker

University of Essex, England
nvoelker@essex.ac.uk

Abstract. The standard versions of HOL only support disjoint sums over finite families of types. This paper introduces disjoint sums over type classes containing possibly a countably infinite number of monomorphic types. The result is a monomorphic sum type together with an overloaded function which represents the family of injections. Model-theoretic reasoning shows the soundness of the construction.

In order to axiomatize the disjoint sums in HOL, datatypes are introduced which mirror the syntactic structure of type classes. The association of a type with its image in the sum type is represented by a HOL function *carrier*. This allows a translation of the set-theoretic axiomatization of disjoint sums to HOL.

As an application, a sum type *U* is presented which contains isomorphic copies of many familiar HOL types. Finally, a *Z* universe is constructed which can server as the basis of a HOL model of the *Z* schema calculus.

1 Introduction

Gordon's version of higher order logic is well-known as the logic underlying the HOL theorem proving assistant [3]. It features a polymorphic type system which was extended in Isabelle/HOL [12] to include overloading controlled by type classes. Although this is quite an expressive type system, there are applications for which it can be too rigid. This has prompted research into the relationship of type theory and untyped set theories [9, 15] and possible combinations of the two [2].

Instead of turning to set theory, this paper tries to extend the expressiveness of HOL by introducing disjoint sums over type classes. The approach applies to syntactic type classes generated from a finite number of type constructors. Given such a class *C*, the disjoint sum consists of a new type *V* together with an overloaded function *emb* which injects each type of *C* into *V*. Furthermore, *V* is the disjoint union of the *emb*-images of the types in *C*.

The existence of isomorphic copies of different types within one type makes constructions possible which are otherwise problematic due to typing restrictions. In particular, it supports the definition of types which can serve as semantical domains for other formalisms. For example, in Section 8, a *Z* universe is constructed. The latter should be useful as the basis of a HOL model of the *Z* schema calculus which is more faithful than previous approaches.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 5{18, 1999.

2 Syntactic Type Classes

In Isabelle, type classes were added to HOL as a way of controlling the instantiation of type variables. This was originally inspired by the syntactic view of type classes in programming languages such as Haskell [6]. Subsequently, axiomatic type classes [16] have been introduced in Isabelle/HOL.

In this paper, only syntactic, non-axiomatic type classes are considered. The membership of a type T to a class C is written as $T :: C$. A class is specified by a finite number of syntactic *arity declarations*. As an example, consider a class N with two arity declarations:[1]:

$$\begin{array}{l} nat :: N \\ fun :: (N ; N) \; N \end{array}$$

The first arity declarations means that the natural number type *nat* is in class N. The second arity declaration says that class N is closed under the formation of function spaces, i.e. given $A :: N$ and $B :: N$, the function space $A \; ! \; B$ is also in class N. Some monomorphic types in N are nat, $nat \; ! \; nat$ and $(nat \; ! \; nat) \; ! \; nat$.

Polymorphic types belonging to a certain class can be formed with the help of type variables restricted to that class. For example, class N contains the restriction $(\;\; :: N)$ of type variable to N. Another example for a polymorphic type in N is $(\;\; :: N) \; ! \; nat$. Class-restricted type variables are the only means to build types in \HOL with type classes" which is not available in \HOL without type classes".

The type constructors *nat* and *fun* generate the monomorphic types in N similar to the way an inductive datatype is generated by its constructors. In view of this analogy, it is tempting to speak of *inductive type classes*.

Compared to Isabelle's (non-axiomatic) type classes, this paper makes a number of simplifying assumptions. These are mainly for presentation reasons.

Note 1. Type classes are assumed to be non-empty and static. Each type class is thus specified by a fixed, finite number of arity declarations. Issues concerning class inclusion and ordering will be disregarded.

Note 2. Type class N is special in so far as its arity declarations do not involve other type classes. Only such homogeneous classes will be considered.

By HOL's type instantiation rule, type variables in a theorem can be instantiated to other types under certain provisos. In HOL with type classes, this rule is amended so that a substitution of a class-restricted type variable has to respect its class, i.e. a substitution of $(\;\; :: N)$ by a type T requires T to be of class N. This amendment of the type instantiation rule is the only change to HOL's basic axioms caused by the introduction of type classes.

[1] The function space type constructor is usually written in infix-form, i.e. $(\; ; \;)fun = (\; ! \;)$

Overloading can be expressed by polymorphic constants whose type contains class-restricted type variables. As an example, the -symbol is declared to be a binary relation on the type class N:

$$:: [\quad :: N;\quad]\ \ !\ \ \mathit{bool}$$

The inductive generation of type classes from type constants and polymorphic type constructors gives rise to a definition principle[2]. For example, the relation can be specified on N by:

$$\begin{array}{ll} 8(a :: \mathit{nat})\ b{:} & a \quad b = (9n{:}b = a + n) \\ 8(f :: (\quad :: N)\ !\ (\quad :: N))\ g{:} & f \quad g = (8x{:}f\ x \quad g\ x) \end{array}$$

Similar to the definition of primitive recursive functions on datatypes, it can be shown that under suitable conditions there exists a unique constant or function which solves such equations [16]. In general this requires as many equations as there are arity declarations for the type class.

For a datatype, structural induction and the principle of primitive recursive function definition can be expressed directly as HOL theorems. This is not possible for HOL classes. Consider for example induction on class N:

$$\begin{array}{l} 8(n :: \mathit{nat}){:}\ P\ n \\ 8(\quad :: N){:}8(\quad :: N){:}\ (8(x :: \quad){:}P\ x) \wedge (8(y :: \quad){:}P\ y)\ \ !\ \ 8(z :: \quad !\quad){:}P\ z \\ \hline 8(\quad :: N){:}8(x :: \quad){:}\ P\ x \end{array}$$

There are two problems with formulating this induction principle as a HOL theorem:

1. In HOL formulas, type variables are always universally quantified at the outermost level. There is no nesting of quantifiers such as in System F [1]. This makes it impossible to express that the second premise has to be true for *all* types $:: N$ and $:: N$.
2. In a HOL theorem, the predicate P would be a (universally quantified) bound variable whose occurrences are of different types. Such polymorphic bound variables are not permitted in HOL. [3]

In Isabelle/HOL, the lack of explicit induction theorems for type classes is compensated mostly by axiomatic type classes. Problems only seem to occur with the current restriction to axioms with at most one type variable. In Section 6 it is shown that the disjoint sum construction can provide alternatives for reasoning with type classes.

The restriction of a polymorphic function $f :: (\quad :: C)\ !\quad$ to some type $T :: C$ is denoted by f_T. Unless otherwise stated explicitly, the term HOL refers in the sequel to higher order logic with syntactic type classes as sketched above.

[2] Called primitive recursion over types in [16]

[3] This is exploited in the Isabelle implementation of terms: the representation of bound variables is untyped.

3 Representation of Type Syntax within HOL

The axiomatization of disjoint sums will make it necessary to construct HOL formulas which concern all monomorphic types in a certain class. As seen in the previous section, this can be difficult due to the limitations of HOL's polymorphism.

The key step in our approach is a representation of the syntax of the monomorphic types in classes by HOL datatypes. In the case of the example class N, a corresponding datatype N^y is defined by:

$$N^y = nat^y \; j \; fun^y \; N^y \; N^y$$

The monomorphic types in class N and the elements of the datatype N^y are obviously in a one-to-one correspondence. The same construction is possible for any type class C. It is assumed that constructor names are chosen appropriately in order to avoid name clashes with existing constants.

Definition 1. *Let C be some type class. Then C^y denotes a datatype which contains for every n-ary type constructor of C a corresponding n-ary datatype constructor ^{y}.*

Proposition 1. *The y-bijection between the type constructors of a class C and the constructors of the associated datatype C^y extends to a bijection between the monomorphic types in C and the elements of C^y.* ⊓⊔

The element of C^y corresponding to some monomorphic type $T :: C$ is denoted by T^y. By associating type variables with HOL variables, it would be possible to extend Proposition 1 to polymorphic types.

Of course, the y-mapping from types to their representation is not a HOL function. However, using primitive recursion over types, it is possible to define an overloaded HOL function *tof* (short for \type of") which takes the elements of the types in C to the representation of their types:

Definition 2. *Let C be some type class with corresponding datatype C^y as in Definition 1. The function*

$$tof :: (\;\; :: C) \;\; ! \;\; C^y$$

is defined by primitive recursion over the type class C as follows:

1. *For a 0-ary type constructor $:: C$:*

$$tof\,(x :: \;\;) \;=\; {}^y$$

2. *and for an n-ary type constructor $:: (C; \ldots)C$ with $n > 0$:*

$$tof\,(x :: (\;_1; \ldots; \;_n) \;) \;=\; {}^y \,(tof\,(arbitrary :: \;_1)) \; \ldots (tof\,(arbitrary :: \;_n))$$

where $(arbitrary :: \;)$ denotes some unspecified element of type .

A concrete definition of (*arbitrary* ::) could be given using the HOL version of Hilbert's -operator, often called \select" or \choice".

Note 3. The function *tof* defined above depends on the class C and the datatype C^y. This could be documented by a subscript, i.e. by writing tof_C. For the sake of better readability, we will refrain here and in the sequel from such indexing as long as the dependencies are clear from the context.

For class N, Definition 2 amounts to:

$$\begin{aligned} tof \ :: \ (\quad :: N) \quad ! \quad & N^y \\ tof \ (x :: nat) \quad = \quad & nat^y \\ tof \ (x :: (\quad ! \quad)) \quad = \quad & fun^y \ (tof \ (arbitrary :: \quad)) \ (tof \ (arbitrary :: \quad)) \end{aligned}$$

This implies for example:

$$tof \ (x :: nat \ ! \ nat) \ = \ fun^y \ nat^y \ nat^y \tag{1}$$

Proposition 2. *Let C, C^y and tof be as in Definition 2. Then for any monomorphic type $T :: C$:*

$$tof \, (x :: T) \ = \ T^y \tag{2}$$

Proof. The statement follows from the definition of the function *tof* and the y-mapping on types by induction over the structure of T. ⊓⊔

Note that (2) is not a HOL formula. This is a consequence of the fact that the association of the type T with its representation T^y is not a HOL function. However, for any specific monomorphic type $T :: C$, the T instance of (2) can be stated and proven in HOL, see for example (1).

Corollary 1. *Assume C, C^y and tof as in Definition 2. Then the function tof is constant on every type:*

$$tof \, (x :: \quad) \ = \ tof \, (y :: \quad) \tag{3}$$

Although (3) is a valid HOL formula, its proof requires structural induction over the types in C. In theorem provers which do not provide this or an equivalent method of proof, the equation can not be established. In this case, it would be possible to add it as an axiom. Of course, as with (2), all monomorphic instances of (3) can be proven easily in HOL.

In order for a type V and an overloaded function $emb :: (\quad :: C) \ ! \ V$ to be the disjoint sum over a type class C, it is necessary that the *emb*-images of the monomorphic types $T :: C$ are disjoint. If the type class C contains an infinite number of (monomorphic) types, then it is not possible to capture this requirement directly by listing all instances. As a way out of this dilemma, the type-indexed set family $(range \; emb_T)_{T::C}$ will be represented by a HOL function named *carrier* which operates on the datatype associated with the class C.

Definition 3. *Let C be some type class with associated datatype $C^\dagger$ and function tof as in Definition 2. Further, let $f :: (\alpha :: C) \to T$ be an overloaded function from the types in C to some type T. The function carrier$_f$ is defined by:*

$$carrier_f \ :: \ C^\dagger \ \to \ T \ set$$
$$carrier_f \ (tof \ (x :: \alpha)) \ = \ range \ f \tag{4}$$

Equation (4) defines the function $carrier_f$ uniquely. This follows from Proposition 2 and the fact that $tof \ (x :: \alpha)$ ranges over all of $C^\dagger$ when α ranges over the types in C.

Applying Proposition 2 to (4) yields for monomorphic types $T :: C$:

$$carrier_f \ T^\dagger = range \ f_T \tag{5}$$

A concrete instance of this equation for the class N and some overloaded function f is:

$$carrier_f \ (fun^\dagger \ nat^\dagger \ nat^\dagger) = range \ f_{nat \to nat} \tag{6}$$

Note that similar to the case of equation (2), equation (5) is not a HOL formula because it contains the term $T^\dagger$. Again, any monomorphic instance can be stated and proven in HOL, see for example (6).

4 Extension of HOL's Semantic Foundation

Disjoint sums are a well-known mechanism for introducing new objects in set theory or category theory. In standard HOL, only the disjoint sum of a finite number of HOL types can be formed. As a first step towards a remedy of this situation, the semantic foundation of HOL is extended.

Recall that the HOL semantics in [3] is based on a universe U of non-empty sets which fulfills a number of closure properties. This includes closure with respect to the forming of non-empty subsets, finite cartesian products and powerset. We add a further axiom to U, namely that it is closed under countable products:

> For any countably infinite family $(A_i)_{i \in \omega}$ of sets in U, the product $\prod_{i \in \omega} A_i$ is also a member of U.

The existence of such a universe U can be shown in ZFC. Just as in the case of the standard HOL universe [3], one could choose U to be the non-empty sets in the von Neumann cumulative hierarchy before stage $\omega + \omega$. The same extension of U was previously used by Regensburger for the construction of inverse limits [13].

Proposition 3. *HOL is sound with respect to the enlarged universe U.*

Proof. (Sketch) It is known that HOL without type classes is sound with respect to any universe which fulfills the closure properties stated in [3]. The result can be extended to HOL with type classes. Essentially, given a model M, the meaning of a term containing a class-restricted type variable :: C is obtained by considering all instantiations of with elements of U which are interpretations in M of a monomorphic type in class C. Adding a further closure requirement on the underlying universe does not invalidate soundness. ⊓⊔

Definition 4. *For a family $(A_i)_{i2I}$ of sets in U, let $_i$ be the projection from the product* Q $_{i2I}$ *A_i to its i'th component A_i. The sum of the family $(A_i)_{i2I}$ is defined by:*

$$\overset{\text{X}}{\underset{i2I}{}} A_i = fxj9i\,2\,I:9a_i\,2\,A_i:x = f(y;j)\,2\,(\overset{\text{Y}}{\underset{i2I}{}}\;A_i)\quad I\;j\;\;_i(y) = a_i \,\hat{}\, j = igg$$

Proposition 4. *The extended universe U is closed under finite and countably infinite sums, i.e. for any finite or countably infinite family $(A_i)_{i2I}$ of sets with I and all A_i in U, the sum* P $_{i2I}$ *A_i is also an element of U.*

Proof. The sum is per definition a non-empty subset of IP ((Q $_{i2I}$ A_i) I). Hence the statement follows from the closure properties of U with respect to finite and countably infinite products, non-empty subsets and the power-set construction. ⊓⊔

5 Disjoint Sums over Type Classes

What does it mean that a type V together with an overloaded function *emb* :: (:: C) $!$ V is a disjoint sum over a type class C? A consideration of set theory suggests the following three requirements:

1. For every monomorphic type T :: C, the mapping emb_T should be injective.
2. The images of different injections should be disjoint, i.e. for all monomorphic types A :: C, B :: C with $A 6= B$: $range\ emb_A\ \backslash\ range\ emb_B = fg$
3. V should be the union of the images of the monomorphic types T of class C, i.e.: $8\,v\,2\,V\!: 9T :: C\!: v\,2\ range\ emb_T$

Unfortunately, not all of these requirements translate directly to HOL formulas. In fact, while the first requirement can be represented easily in HOL by the simple formula *inj emb*, it is not clear how the remaining two requirements can be formulated in HOL. After all, neither the inequality of two HOL types nor the existence of a certain HOL type are HOL formulas. Note in particular that the formula:

$$8v\!: v\,2\ range\ emb$$

is *not* a valid formulation of the requirement that every element v :: V of the disjoint sum type V lies in the range of emb_T for some type T :: C. Instead, due to the implicit universal quantification over type variables in HOL, this formula would demand every element v :: V to lie in the range of all mappings emb_T

for $T :: C$. For type classes C containing two or more types, this contradicts the disjointness requirement.

The two problematic disjoint sum requirements put conditions on the sets ($range\ emb_T$) where T is a monomorphic type of class C. These conditions become expressible in HOL if the indexing of these sets by types is replaced by an indexing with the elements of the datatype associated with C. This was the reason for the introduction of the function *carrier* in Section 3. With the help of this function, the characterization of disjoint sums in HOL becomes straightforward.

Definition 5. *Let C be some type class and $emb :: (\alpha :: C) \to V$ an overloaded function from the types in C to some type V. Let $C^\dagger$, tof and*

$$carrier = carrier_{emb}$$

be as in Definition 1, 2 and 3. Then (V, emb) is a disjoint sum over class C provided the following three HOL formulas are valid:

$$inj\ emb \tag{7}$$

$$\forall A\ B.\ A \neq B \longrightarrow carrier\ A \cap carrier\ B = \{\} \tag{8}$$

$$\forall x.\ \exists T.\ x \in carrier\ T \tag{9}$$

Proposition 5. *Let (V, emb) be a disjoint sum over a type class C. Then V and emb fulfill the three requirements for disjoint sums stated above.*

Proof. The statement follows immediately from the definition of $C^\dagger$, *tof* and *carrier* and equation (5). $\square$

The existence of disjoint sums over type classes follows from the existence of sums in the (extended) underlying set model of HOL:

Proposition 6. *Let C be a non-empty type class in some HOL theory $\mathcal{T}$. Then it is possible to extend $\mathcal{T}$ by a new type V and an overloaded mapping $emb :: (\alpha :: C) \to V$ such that V is the disjoint sum of the types in C with embedding emb. Furthermore, if the original theory $\mathcal{T}$ has a standard model, then this can be extended to a standard model of the extended theory.*

Proof. (*Sketch.*) Choose the name of the new type V and the overloaded constant *emb* in such a way that clashes with the names of existing types and constants are avoided. Assume a standard model M of the theory $\mathcal{T}$ including meanings of the class C and the constants *tof* and *carrier*. The family of meanings $M(T)$ of monomorphic types $T :: C$ forms a countable family of sets in U. According to Proposition 4, the disjoint sum S over this family exists in U. A meaning for the type V and the constant *emb* is given in M by associating: (i) V with the sum S and (ii) $emb :: (\alpha :: C) \to V$ with a (dependently typed) function which given a set $M(T)$ yields the inclusion function from that set into S. The latter is justified because each possible meaning of a type variable $\alpha :: C$ is of the form $M(T)$ for some monomorphic type $T :: C$. The validity of the conditions (7) - (9) in the model M follows from (5) and Definition 4. $\square$

Since the existence of a standard model implies consistency, it follows that the extension of theories by disjoint sums over type classes does not compromise consistency. Note that the disjoint sum type V itself is not a member of the class C. Hence, there is no self-reference in the construction of V. This is reminiscent of the extension of the Calculus of Constructions by Σ-types in [10]: the predicativity makes the theory extension safe and allows the formulation of a set-theoretic semantics.

As an extreme case of an application of Proposition 6, let $C_{\mathtt{T}}$ be the class of all types which can be formed in some HOL theory $\mathtt{T}$. Then the proposition justifies the extension of $\mathtt{T}$ by a “universe” $V_{\mathtt{T}}$ which is the disjoint sum of all monomorphic types in $\mathtt{T}$.

The difficulties with the HOL formulation are not intrinsic to the set-theoretic characterization of disjoint sums. Suppose one tried instead a category-theory inspired axiomatization which works on the level of functions. In this approach, one would replace the three disjoint sum requirements above by the single requirement that for every overloaded function $g :: (\alpha :: C) \to \tau$, there exists a unique function $f :: V \to \tau$ with $g = f \circ \mathit{emb}$. Alas, a direct HOL formulation of this requirement leads to the same two problems of missing nested quantification and disallowed polymorphic variables which hindered previously the HOL formulation of the induction principle for type classes on page 7.

It should be stressed that other axiomatizations of disjoint sums over type classes in HOL are possible. For example, one could postulate the existence of suitable constants which permit a primitive recursive definition of the *carrier* function. The axiomatization presented above aims to be a relatively straightforward translation of the usual set-theoretic formulation.

6 First Developments

In the following, let $(V; \mathit{emb})$ be a disjoint sum over a type class C with associated datatype $C^\dagger$ and functions *tof* and *carrier* as in Definition 5. Since the carrier sets partition the type V, a function *type_of* can be defined which associates each element $v :: V$ with the representation of the type in whose carrier it lies:

$$\begin{aligned} \mathit{type\_of} &:: V \to C^\dagger \\ \mathit{type\_of}\ v &= (\varepsilon T.\ v \in \mathit{carrier}\ T) \end{aligned}$$

The ε denotes Hilbert's choice operator. Using the rules (8) and (9), the following properties of *type_of* can be proven easily in HOL:

$$\begin{aligned} (x \in \mathit{carrier}\ T) &= (\mathit{type\_of}\ x = T) \qquad (10) \\ \mathit{type\_of}\ (\mathit{emb}\ x) &= \mathit{tof}\ x \end{aligned}$$

Equivalence (10) implies further:

$$x \in \mathit{carrier}\ (\mathit{type\_of}\ x)$$

From the induction theorem of the datatype C^y, an induction theorem for V based on the function *carrier* can be derived. The theorem is stated here only for the example type class N:

$$\begin{array}{l} [j\ 8a\ 2\ carrier\ nat^y\!: P\ a; \\ \quad 8A\ B\ c\!: [j\ c\ 2\ carrier\ (fun^y\ A\ B); \\ \qquad\qquad 8a\ 2\ carrier\ A\!: P\ a; \\ \qquad\qquad 8b\ 2\ carrier\ B\!: P\ b\ j]\ \ !\ \ P\ c \\ j]\ \ !\ \ P\ x \end{array}$$

The functions *tof* and *type_of* are an example of a more general correspondence between overloaded functions on class C and functions on the disjoint sum type V. Given any overloaded function $f :: (\ \ :: C)\ !\ $, the axioms of V imply the existence of a unique function $g :: V\ !\ $ such that:

$$f = g \quad emb \tag{11}$$

Conversely, given any function $g :: V\ !\ $, the corresponding overloaded function $f :: (\ \ :: C)\ !\ $ is trivially determined by (11).

This correspondence of functions can be used for example to give a simple definition of a equality relation *eq* across types in class N:

$$\begin{array}{ll} eq & :: [\ \ :: N;\ \ :: N]\ \ !\ \ bool \\ x\ eq\ y = (emb\ x = emb\ y) & \end{array}$$

Alternatively, the relation *eq* could be defined by primitive recursion over the types in N. However, proving properties of *eq* such as symmetry:

$$(x\ eq\ y) = (y\ eq\ x)$$

based on the primitive recursive definition requires some sort of structural induction over type classes, be it in the form of axiomatic types with axioms which are parameterized by two type variables. In comparison, the above definition of *eq* via the disjoint sum type makes the proof of symmetry trivial.

7 A Universe for Many HOL Types

Before the construction of the universe, it is convenient to introduce some terminology.

Definition 6. *Let A and B be HOL types. Then we say that A can be embedded in B, provided there exists an injection from A to B.*

Note that the definition applies both to monomorphic and polymorphic HOL types.

Definition 7. *Let be a type constructor of some arity n 0. Then we say that is monotonic, if given n injections $f_1 :: A_1\ !\ B_1; : : : ; f_n :: A_n\ !\ B_n$, there exists an injection from $(A_1; : : : ; A_n)$ into $(B_1; : : : ; B_n)$.*

As a special case of Definition 7, 0-ary type constructors are also monotonic.

Proposition 7. *The function space type constructor is monotonic.*

Proof. The statement follows from the HOL theorem:

$$inj\ \ f\ \wedge\ inj\ \ g\ \ !\ \ inj\ \ (\quad h{:}\ g\quad h\quad f^{-1})$$

Proposition 8. *Datatype type constructors are monotonic.*

Proof. Every datatype type constructor can be associated with a function *map* which is functorial in all arguments, i.e.:

$$\begin{aligned} map\quad & \underbrace{id\ ;:::;\ id}_{n} && = \ id \\ map\quad & (f_1\quad g_1)\ :::\ (f_n\quad g_n) && = \ map\quad f_1\ :::\ f_n\quad map\quad g_1\ :::\ g_n \end{aligned}$$

Hence the injectivity of $f_1; :::; f_n$ implies the injectivity of *map* $f_1\ :::\ f_n$. ⊓⊔

Proposition 9. *Let C be some type class containing only monotonic type constructors and let D be the extension of C by an n-ary monotonic type constructor . Let $_1; :::;\ _n$ be n distinct type variables and assume that $(\ _1; :::;\ _n)$ can be embedded in some type of class C. Then every type T in class D can be embedded in a type of class C*

Proof. Suppose first that is of arity 0. Then the statement follows from the monotonicity of the type constructors by induction over the structure of type T. This leaves the case of a polymorphic type constructor of arity $n > 0$. The proof is again by induction on the structure of T. For the induction step, the case of a type T with an outermost type constructor unequal follows by monotonicity. This leaves the case of $T = (A_1; :::; A_n)$ where the types $A_1; :::; A_n$ can be embedded in types $B_1; :::; B_n$ of class C. By monotonicity of , T can be embedded in $T^0 = (B_1; :::; B_n)$. Since $(\ _1; :::;\ _n)$ can be embedded in some type D of class D, the instantiation of that embedding to T^0 provides an embedding of T^0 into an instantiation of D which is of class C. ⊓⊔

Let H be the type class generated by the three basic, monotonic type constructors *bool*, *fun* and *ind*. Most type constructors in the libraries coming with the HOL system or Isabelle/HOL are monotonic and are introduced by an embedding into an already existing type or as a type abbreviation. It follows by induction from Proposition 9 that many HOL types used in practice can be embedded into a type of class H. In particular, this includes all types which can be formed from the type constructors *bool*, *ind*, *fun*, *set* and datatypes such as lists or binary trees. Furthermore, if $(U; emb)$ is a disjoint sum over H, then all these types can even be embedded in the single monomorphic type U.

In many applications of HOL, it is possible to restrict the type system to types which can be embedded in U. In this case, the type U can serve as a universe which supports generic constructions such as the definition of datatypes and co-datatypes. This then alleviates the need for special universes such as the one employed by Paulson in his construction of HOL datatypes [11].

8 A Semantical Domain for *Z*

Embedding a specification formalism such as *Z* [14] in higher order logic is attractive because of the automation provided by theorem prover tools like the HOL system and Isabelle/HOL. In order to keep the embedding simple, it is advisable to reuse HOL's infrastructure as much as possible. For typed formalisms, this means that its typing system should ideally be represented by that of HOL. This is a problem in the case of *Z* as it would require record types not present in HOL.

Previous work [7, 8] on *Z* embeddings has side-stepped this problem by encoding schemas before analyzing them in HOL. This resulted in shallow embeddings well suited for reasoning about individual specifications. However, the encoding made it difficult to derive general theorems about the *Z* schema calculus itself. It also causes some duplication in proof efforts.

As an alternative, we will represent *Z* values as elements of a single, monomorphic type *ZU*. While this means that *Z* typing has to be performed explicitly by induction in HOL, it provides a basis for a more faithful model of the *Z* schema calculus in HOL.

Following [5], we consider *Z* to be a typed set theory with natural numbers as the only primitive type. Type forming operations are sets, binary products and records. In the context of *Z*, the latter are often called schemas. As a first step, the syntactical type structure of *Z* is represented by a datatype *ZT*:

$$ZT = NatT \ j \ SetT \ \ ZT \ j \ PrdT \ \ ZT \ \ ZT \ j \ RcdT \ \ (ZT \ \ rcd) \qquad (12)$$

Here *rcd* is a 1-ary type constructor of type-homogeneous records. In other words, *rcd* is the type of all finite mappings from some type of identifiers to . Such a type constructor can be defined easily in HOL via a bijection with the set of all finite, univalent relations of type (*string*)*set*. Because of the finiteness, the recursion over this type constructor in (12) poses no problem, c.f. [11].

In order to apply the disjoint sum construction, a class is required which contains a - not necessarily unique - representation of every *Z* value. As a first attempt, we consider a class Z_0 with arity declarations which follow the structure of the datatype *ZT*:

$$\begin{array}{l} nat :: Z_0 \\ set \ :: (Z_0) \ Z_0 \\ prd :: (Z_0; Z_0) \ Z_0 \\ rcd :: (Z_0) \ Z_0 \end{array}$$

Unfortunately, the resulting class Z_0 does *not* contain a faithful representation of all *Z* values. This is because the fields of a record in Z_0 are all of the same type, while for *Z* schemas, there is no such restriction.

Instead, our approach relies on an encoding of records as labelled binary products. The corresponding type constructor *lprd* is:

$$lprd \ \ \approx \ string$$

The empty record is represented as the only element () of the one-element type *unit*. As an example, here is the representation of a record r which consists of two fields \\$h$" and \\$p$" associated with the values 1 and $(2;3)$:

$$r \;=\; < \backslash h\text{"} = 1; \backslash p\text{"} = (2;3) > \;\; \cong \;\; (\backslash h\text{"}; 1; (\backslash p\text{"}; (2;3); ()))$$

The encoding of records is made unique by requiring that labels occur at most once and are sorted in lexicographical order. Furthermore, in order to distinguish empty sets of the same HOL type but different Z type, sets will be labelled with their Z type. This relies on a type constructor *tset* such that:

$$tset \cong \quad set \qquad ZT$$

The construction of the Z universe is based on the following type class Z:

$$\begin{array}{l}
nat \;:: Z \\
tset \;:: (Z)\; Z \\
prd \;:: (Z;Z)\; Z \\
unit :: Z \\
lprd :: (Z;Z)\; Z
\end{array}$$

While every Z value has a unique representation in this class, there are elements of Z which do not represent a Z value:

1. Z contains labelled products with duplicate or unsorted labels.
2. Z contains sets labelled with a Z type which does not agree with the Z type of all its arguments.

Let $(V; emb)$ be a disjoint sum over Z. Then the spurious elements can be filtered out with the help of appropriate predicates on V. The remaining subset of V still contains a unique representation of every Z value. Introducing a new type ZU via an isomorphism with this subset yields a semantic domain for Z in HOL.

9 Concluding Remarks

Disjoint sums are a familiar construction in set theory. By representing inclusions as overloaded functions, it becomes feasible to add disjoint sums over type classes to HOL. From a model-theoretic point of view, this extension is straightforward. In particular, it does not compromise consistency. The key to a HOL axiomatization of the sum types lies in mirroring the type structure of classes by HOL datatypes. This then allows a HOL definition of the carrier of a type. Using this function, the set-theoretic characterization of disjoint sums can be translated easily to HOL. A similar development should be possible for products over type classes.

Two primary applications of the disjoint sum types are the support for generic constructions such as datatypes and embeddings of formalisms with powerful type systems. The latter was sketched for the case of the specification language

Z. It would be interesting to investigate the usefulness of the approach for other generic constructions such as Scott's inverse limits.

Acknowledgments. Most of the work described in this paper was carried out as part of the project \Implementing Constructive Z", EPSRC research grant GR/L57913. Elwood Wilkins, Ray Turner and Martin Henson have provided valuable feedback on previous versions of this paper. The anonymous referees deserve thanks for their constructive criticism.

References

[1] J.-Y. Girard. The system *F* of variable types, fifteen years later. *Theoretical Computer Science*, 45:159{192, 1986.

[2] M.J.C. Gordon. Set Theory, Higher Order Logic or Both? In Gunter and Felty [4], pages 191{201.

[3] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[4] E. L. Gunter and A. Felty, editors. *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, LNCS 1275. Springer-Verlag, 1997.

[5] M.C. Henson and S. Reeves. A logic for the schema calculus. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 172{191. Springer-Verlag, 1998.

[6] S.P. Jones and J. Hughes, editors. *HASKELL98: A Non-strict, Purely Functional Language*, February 1999. On-line available via `http://haskell.org/`.

[7] J.P.Bowen and M.J.C. Gordon. Z and HOL. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141{167. Springer-Verlag, 1994.

[8] Kolyang, T. Santen, and B. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics | 9th International Conference*, LNCS 1125, pages 283{298. Springer Verlag, 1996.

[9] L. Lamport and L.C. Paulson. Should your specification language be typed? Technical Report 425, Computer Laboratory, University of Cambridge, May 1997.

[10] Zhaohui Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1):107{137, 1991.

[11] L.C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction | CADE-12*, LNAI 814, pages 148{161. Springer-Verlag, 1994.

[12] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer-Verlag, 1994.

[13] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. Dissertation an der Technischen Universität München, 1994.

[14] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988.

[15] R. Turner. Sets, types and typechecking. *Journal of Logic and Computation*, To Appear.

[16] M. Wenzel. Type classes and overloading in higher-order logic. In Gunter and Felty [4], pages 307{322.

Inductive Datatypes in HOL { Lessons Learned in Formal-Logic Engineering

Stefan Berghofer and Markus Wenzel

Technische Universität München
Institut für Informatik, Arcisstraße 21, 80290 München, Germany
`http://www.in.tum.de/ berghofe/`
`http://www.in.tum.de/ wenzelm/`

Abstract. Isabelle/HOL has recently acquired new versions of definitional packages for inductive datatypes and primitive recursive functions. In contrast to its predecessors and most other implementations, Isabelle/HOL datatypes may be mutually and indirect recursive, even infinitely branching. We also support inverted datatype definitions for characterizing existing types as being inductive ones later. All our constructions are fully definitional according to established HOL tradition. Stepping back from the logical details, we also see this work as a typical example of what could be called \Formal-Logic Engineering". We observe that building realistic theorem proving environments involves further issues rather than pure logic only.

1 Introduction

Theorem proving systems for higher-order logics, such as HOL [5], Coq [4], PVS [15], and Isabelle [18], have reached a reasonable level of maturity to support non-trivial applications. As an arbitrary example, consider Isabelle/Bali [14], which is an extensive formalization of substantial parts of the Java type system and operational semantics undertaken in Isabelle/HOL.

Nevertheless, the current state-of-the-art is not the final word on theorem proving technology. Experience from sizable projects such as Isabelle/Bali shows that there are quite a lot of requirements that are only partially met by existing systems. Focusing on the actual core system only, and ignoring further issues such as user interfaces for theorem provers, there are several layers of concepts of varying logical status to be considered. This includes purely syntactic tools (parser, pretty printer, macros), type checking and type inference, basic deductive tools such as (higher-order) unification or matching, proof procedures (both simple and automatic ones), and search utilities | just to name a few.

Seen from a wider perspective, the actual underlying logic (set theory, type theory etc.) becomes only one element of a much larger picture. Consequently, making a theorem proving system a \success" involves more than being good in the pure logic rating. There is a big difference of being able to express certain concepts *in principle* in some given logic vs. offering our customers scalable mechanisms for *actually doing* it in the system.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 19{36, 1999.

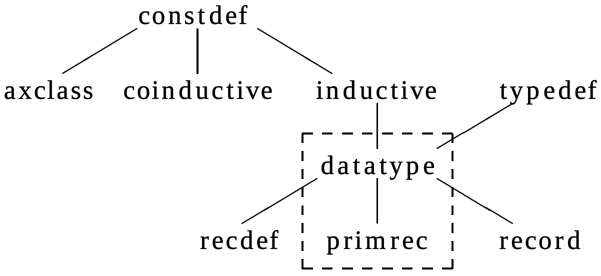
Advanced *definitional mechanisms* are a particularly important aspect of any realistic formal-logic environment. While working in the pure logic would be sufficient in principle, actual applications demand not only extensive libraries of derived concepts, but also general mechanisms for introducing certain kinds of mathematical objects. A typical example of the latter would be inductive sets and types, together with recursive function definitions.

According to folklore, theorem proving is similar to programming, but slightly more difficult. Apparently, the same holds for the corresponding development tools, with an even more severe gap of sophistication, though. For example, consider the present standard in interactive theorem proving technology related to that of incremental compilers for languages such as ML or Haskell. Apparently, our theorem provers are still much more primitive and inaccessible to a wider audience than advanced programming language compilers. In particular, definitional mechanisms, which are in fact resembling a “theory compiler” quite closely, are often much less advanced than our users would expect.

An obvious way to amend for this, we argue, would be to transfer general concepts and methodologies from the established disciplines of Software and Systems Engineering to that of theorem proving systems, eventually resulting in what could be called *Formal-Logic Engineering*.

Getting back to firm grounds, and the main focus of this paper, we discuss the new versions of advanced definitional mechanisms that Isabelle/HOL has acquired recently: **inductive** or **coinductive** definitions of sets (via the usual Knaster-Tarski construction, cf. [17]), inductive datatypes, and primitive recursive functions. Our primary efforts went into the **datatype** and **primrec** mechanisms [2], achieving a considerably more powerful system than had been available before. In particular, datatypes may now involve *mutual* and *indirect recursion*, and *arbitrary branching* over existing types.[1] Furthermore, datatype definitions may now be *inverted* in the sense that existing types (such as natural numbers) may be characterized later on as being inductive, too.

The new packages have been designed for cooperation with further subsystems of Isabelle/HOL already in mind: **recdef** for general well-founded functions [21, 22], and **record** for single-inheritance extensible records [13]. Unquestionably, more such applications will emerge in the future. The hierarchy of current Isabelle/HOL definitional packages is illustrated below. Note that **constdef** and **typedef** refer to HOL primitives [5], and **axclass** to axiomatic type classes [24].



[1] Arbitrary (infinite) branching is not yet supported in Isabelle98-1.

The basic mode of operation of any \advanced" definitional package such as datatype is as follows: given a simple description of the desired result theory by the user, the system automatically generates a sizable amount of characteristic theorems and derived notions underneath. There are different approaches, stemming from different logical traditions, of how this is achieved exactly. These approaches can be roughly characterized as follows.

Axiomatic The resulting properties are generated syntactically only, and introduced into the theory as *axioms* (e.g. [16]).

Inherent The underlying *logic is extended* in order to support the desired objects in a very direct way (e.g. [20]).

Definitional Taking an existing logic for granted, the new objects are represented in terms of existing concepts, and the desired properties are *derived from the definitions* within the system (e.g. [2]).

Any of these approaches have well-known advantages and disadvantages. For example, the definitional way is certainly a very hard one, demanding quite a lot of special purpose theorem proving work of the package implementation. On the other hand, it is possible to achieve a very high quality of the resulting system | both in the purely logical sense meaning that no \wrong" axioms are asserted and in a wider sense of theorem proving system technology in general.

The rest of this paper is structured as follows. Section 2 presents some examples illustrating the user-level view of Isabelle/HOL's new datatype and primrec packages. Section 3 briefly reviews formal-logic preliminaries relevant for our work: HOL basics, simple definitions, inductive sets. Section 4 describes in detail the class of admissible datatype specifications, observing fundamental limitations of classical set theory. Section 5 recounts techniques for constructing mutually and indirectly recursive, infinitely branching datatypes in HOL, including principles for induction and recursion. Section 6 discusses some issues of integrating the purely-logical achievements into a scalable working environment.

2 Examples

As our first toy example, we will formalize some aspects of a very simple functional programming language, consisting of arithmetic and boolean expressions formalized as types aexp and bexp (parameter is for program variables).

```
datatype   aexp  =  If (  bexp) (  aexp) (  aexp)
                 j  Sum (  aexp) (  aexp)
                 j  Var
                 j  Num nat
and        bexp  =  Less (  aexp) (  aexp)
                 j  And (  bexp) (  bexp)
```

This specification emits quite a lot of material into the current theory context, first of all injective functions Sum :: aexp ! aexp ! aexp etc. for any of the datatype constructors. Each valid expression of our programming language is denoted by a well-typed constructor-term. Functions on inductive types are typically defined by primitive recursion. We now define evaluation functions for arithmetic and boolean expressions, depending on an environment $e :: \alpha \rightarrow \mathrm{nat}$.

consts
  evala :: $(\alpha \rightarrow \mathrm{nat}) \rightarrow \alpha\ \mathrm{aexp} \rightarrow \mathrm{nat}$
  evalb :: $(\alpha \rightarrow \mathrm{nat}) \rightarrow \alpha\ \mathrm{bexp} \rightarrow \mathrm{bool}$
primrec
  evala e (If $b\ a_1\ a_2$) = (if evalb $e\ b$ then evala $e\ a_1$ else evala $e\ a_2$)
  evala e (Sum $a_1\ a_2$) = evala $e\ a_1$ + evala $e\ a_2$
  evala e (Var v) = $e\ v$
  evala e (Num n) = n
  evalb e (Less $a_1\ a_2$) = (evala $e\ a_1$ < evala $e\ a_2$)
  evalb e (And $b_1\ b_2$) = (evalb $e\ b_1 \wedge$ evalb $e\ b_2$)

Similarly, we may define substitution functions for expressions. The mapping $s :: \alpha \rightarrow \beta\ \mathrm{aexp}$ given as a parameter is lifted canonically on aexp and bexp.

consts
  substa :: $(\alpha \rightarrow \beta\ \mathrm{aexp}) \rightarrow \alpha\ \mathrm{aexp} \rightarrow \beta\ \mathrm{aexp}$
  substb :: $(\alpha \rightarrow \beta\ \mathrm{aexp}) \rightarrow \alpha\ \mathrm{bexp} \rightarrow \beta\ \mathrm{bexp}$
primrec
  substa s (If $b\ a_1\ a_2$) = If (substb $s\ b$) (substa $s\ a_1$) (substa $s\ a_2$)
  substa s (Sum $a_1\ a_2$) = Sum (substa $s\ a_1$) (substa $s\ a_2$)
  substa s (Var v) = $s\ v$
  substa s (Num n) = Num n
  substb s (Less $a_1\ a_2$) = Less (substa $s\ a_1$) (substa $s\ a_2$)
  substb s (And $b_1\ b_2$) = And (substb $s\ b_1$) (substb $s\ b_2$)

The relationship between substitution and evaluation can be expressed by:

lemma
  evala e (substa $s\ a$) = evala ($\lambda x.$ evala e ($s\ x$)) $a \wedge$
  evalb e (substb $s\ b$) = evalb ($\lambda x.$ evala e ($s\ x$)) b

We can prove this theorem by straightforward reasoning involving *mutual* structural induction on a and b, which is expressed by the following rule:

$$\frac{\begin{array}{ll} \forall b\ a_1\ a_2.\ Q\ b \wedge P\ a_1 \wedge P\ a_2 & \Longrightarrow\ P\ (\mathrm{If}\ b\ a_1\ a_2) \\ \vdots & \\ \forall a_1\ a_2.\ P\ a_1 \wedge P\ a_2 & \Longrightarrow\ Q\ (\mathrm{Less}\ a_1\ a_2) \\ \vdots & \end{array}}{P\ a \wedge Q\ b}$$

As a slightly more advanced example we now consider the type (α, β, γ)tree, which is made arbitrarily branching by nesting an appropriate function type.

datatype (α, β, γ)tree = Atom α | Branch β $(\gamma \rightarrow (\alpha, \beta, \gamma)\mathrm{tree})$

Here α stands for leaf values, β for branch values, γ for subtree indexes. It is important to note that γ may be any type, including an infinite one such as nat; it need not even be a datatype. The induction rule for (α, β, γ)tree is

$$\frac{\forall a.\ P\ (\mathrm{Atom}\ a) \quad \forall b\ f.\ (\forall x.\ P\ (f\ x)) \Longrightarrow P\ (\mathrm{Branch}\ b\ f)}{P\ t}$$

Note how we may assume that the predicate P holds for all values of f, all subtrees, in order to show P (Branch b f). Using this induction rule, Isabelle/HOL automatically proves the existence of combinator tree-rec for primitive recursion:

tree-rec :: (!) ! (! (y ! (; ;y)tree) ! (y !) !) ! (; ;y)tree !
tree-rec f_1 f_2 (Atom a) = f_1 a
tree-rec f_1 f_2 (Branch b f) = f_2 b f ((tree-rec f_1 f_2) f)

In the case of Branch, the function tree-rec f_1 f_2 is recursively applied to all function values of f, i.e. to all subtrees. As an example primitive recursive function on type tree, consider the function member c which checks whether a tree contains some Atom c. It could be expressed as tree-rec (a: a = c) (b f f^0: 9x: f^0 x). Isabelle/HOL's primrec package provides a more accessible interface:

primrec
  member c (Atom a) = (a = c)
  member c (Branch b f) = (9x: member c (f x))

3 Formal-Logic Preliminaries

3.1 The Logic of Choice?

This question is a rather subtle one. Actually, when it comes to real applications within a large system developed over several years, there is not much choice left about the underlying logic. Changing the very foundations of your world may be a very bad idea, if one cares for the existing base of libraries and applications.

HOL [5], stemming from Church's \Simple Theory of Types" [3] has proven a robust base over the years. Even if simplistic in some respects, HOL proved capable of many sophisticated constructions, sometimes even *because* of seeming weaknesses. For example, due to simple types HOL admits interesting concepts such as intra-logical overloading [24] or object-oriented features [13]. Our constructions for inductive types only require plain simply-typed set theory, though.

3.2 Isabelle/HOL { Simply-Typed Set Theory

The syntax of HOL is that of simply-typed -calculus. *Types* are either variables , or applications ($_1$;:::; $_n$)t, including function types $_1$! $_2$ (right associative in x). *Terms* are either typed constants c or variables x , applications (t u) or abstractions x:t. Terms have to be well-typed according to standard rules. *Theories* consist of a signature of types and constants, and axioms. Any theory induces a set of derivable theorems ‘ ’, depending on a xed set of deduction rules that state several \obvious" facts of classical set theory. Starting from a minimalistic basis theory, all further concepts are developed *de nitionally*.

Isabelle/HOL provides many standard notions of classical set-theory. Sets are of type set; in x f \ A refers to the image, vimage f A to the reverse image of f on A; inv f inverts a function; lfp F and gfp F are the least and greatest xpoints of F on the powerset lattice. The sum type + has constructors Inl and Inr. Most other operations use standard mathematical notation.

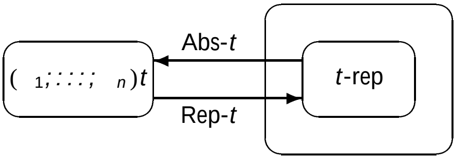
3.3 Simple Definitions

The HOL methodology dictates that only *definitional* theory extension mechanisms may be used. HOL provides two primitive mechanisms: *constant definitions* and *type definitions* [5], further definitional packages are built on top.

Constant definition We may add a new constant c to the signature and introduce an axiom of the form $\vdash c\ v_1 \ldots v_n \equiv t$, provided that c does not occur in t, $TV(t) \subseteq TV(c)$ and $FV(t) \subseteq \{v_1, \ldots, v_n\}$.

Type definition Let $t\text{-rep}$ be a term of type α set describing a non-empty set, i.e. $\vdash u \in t\text{-rep}$ for some u. Moreover, require $TV(t\text{-rep}) \subseteq \{\alpha_1, \ldots, \alpha_n\}$. We may then add the type $(\alpha_1, \ldots, \alpha_n)t$ and the following constants

$$\begin{aligned} \text{Abs-}t &:: \alpha \Rightarrow (\alpha_1, \ldots, \alpha_n)t \\ \text{Rep-}t &:: (\alpha_1, \ldots, \alpha_n)t \Rightarrow \alpha \end{aligned}$$



to the signature and introduce the axioms

$$\begin{aligned} &\vdash \text{Abs-}t\ (\text{Rep-}t\ x) = x && (\textit{Rep-t-inverse}) \\ &\vdash y \in t\text{-rep} \Longrightarrow \text{Rep-}t\ (\text{Abs-}t\ y) = y && (\textit{Abs-t-inverse}) \\ &\vdash \text{Rep-}t\ x \in t\text{-rep} && (\textit{Rep-t}) \end{aligned}$$

Type definitions are a slightly peculiar feature of HOL. The idea is to represent new types by subsets of already existing ones. The axioms above state that there is a bijection (isomorphism) between the set t-rep and the new type $(\alpha_1, \ldots, \alpha_n)t$. This is justified by the standard set-theoretic semantics of HOL [5].

3.4 Inductive Definitions

An **inductive** [17] definition specifies the *least* set closed under certain *introduction rules* — generally, there are many such closed sets. Essentially, an inductively defined set is the least fixpoint $\text{lfp}\ F$ of a certain monotone function F, where $\text{lfp}\ F = \bigcap \{x \mid F\ x \subseteq x\}$. The *Knaster-Tarski* theorem states that $\text{lfp}\ F$ is indeed a fixpoint and that it is the least one, i.e. $F\ (\text{lfp}\ F) = \text{lfp}\ F$ and

$$\frac{F\ P \subseteq P}{\text{lfp}\ F \subseteq P} \qquad \frac{F\ (\text{lfp}\ F \cap P) \subseteq P}{\text{lfp}\ F \subseteq P}$$

where P is the set of all elements satisfying a certain predicate. Both rules embody an induction principle for the set $\text{lfp}\ F$. The second (stronger) rule is easily derived from the first one, because F is monotone. See [17] for more detailes on how to determine a suitable function F from a given set of introduction rules. When defining several *mutually inductive sets* $S_1, \ldots, S_n$, one first builds the sum T of these and then extracts sets S_i from T with the help of the inverse image operator vimage, i.e. $S_i = \text{vimage in}_i\ T$, where in_i is a suitable injection.

4 Datatype Specifications

4.1 General Form

A general **datatype** specification in Isabelle/HOL is of the following form:

$$\begin{array}{llll}\textbf{datatype} & (\alpha_1,\ldots,\alpha_h)t_1 & = & C^1_1\ \tau^1_{1,1}\ \ldots\ \tau^1_{1,m^1_1}\ \mid\ \ldots\ \mid\ C^1_{k_1}\ \tau^1_{k_1,1}\ \ldots\ \tau^1_{k_1,m^1_{k_1}} \\ & \vdots & & \\ \textbf{and} & (\alpha_1,\ldots,\alpha_h)t_n & = & C^n_1\ \tau^n_{1,1}\ \ldots\ \tau^n_{1,m^n_1}\ \mid\ \ldots\ \mid\ C^n_{k_n}\ \tau^n_{k_n,1}\ \ldots\ \tau^n_{k_n,m^n_{k_n}}\end{array}$$

where α_i are type variables, constructors C^j_i are distinct, and $\tau^j_{i,i'}$ are admissible types containing at most the type variables $\alpha_1,\ldots,\alpha_h$. Some type $\tau^j_{i,i'}$ occurring in such a specification is *admissible* iff $\{(\alpha_1,\ldots,\alpha_h)t_1,\ldots,(\alpha_1,\ldots,\alpha_h)t_n\} \vdash \tau^j_{i,i'}$ where $\vdash$ is inductively defined by the following rules:

non-recursive occurrences: $\qquad \Gamma \vdash \sigma$
where σ is non-recursive, i.e. σ does not contain any of the newly defined type constructors $t_1,\ldots,t_n$
recursive occurrences: $\qquad \{\tau\} \cup \Gamma \vdash \tau$
nested recursion involving function types:

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \sigma \to \tau} \qquad \text{where } \sigma \text{ is non-recursive}$$

nested recursion involving existing datatypes:

$$\frac{\begin{array}{c}\{(\sigma'_1,\ldots,\sigma'_{\hat h})\hat t_1,\ldots,(\sigma'_1,\ldots,\sigma'_{\hat h})\hat t_{\hat n}\} \cup \Gamma \vdash \tilde\tau^1_{1,1}[\sigma'_1/\beta_1,\ldots,\sigma'_{\hat h}/\beta_{\hat h}] \\ \vdots \\ \{(\sigma'_1,\ldots,\sigma'_{\hat h})\hat t_1,\ldots,(\sigma'_1,\ldots,\sigma'_{\hat h})\hat t_{\hat n}\} \cup \Gamma \vdash \tilde\tau^{\hat n}_{\hat k_{\hat n},\hat m^{\hat n}_{\hat k_{\hat n}}}[\sigma'_1/\beta_1,\ldots,\sigma'_{\hat h}/\beta_{\hat h}]\end{array}}{\Gamma \vdash (\sigma'_1,\ldots,\sigma'_{\hat h})\hat t_{j'}}$$

where $\hat t_{j'}$ is the type constructor of an existing datatype specified by

$$\begin{array}{llll}\textbf{datatype} & (\beta_1,\ldots,\beta_{\hat h})\hat t_1 & = & D^1_1\ \tilde\tau^1_{1,1}\ \ldots\ \tilde\tau^1_{1,\hat m^1_1}\ \mid\ldots\mid\ D^1_{\hat k_1}\ \tilde\tau^1_{\hat k_1,1}\ \ldots\ \tilde\tau^1_{\hat k_1,\hat m^1_{\hat k_1}} \\ & \vdots & & \\ \textbf{and} & (\beta_1,\ldots,\beta_{\hat h})\hat t_{\hat n} & = & D^{\hat n}_1\ \tilde\tau^{\hat n}_{1,1}\ \ldots\ \tilde\tau^{\hat n}_{1,\hat m^{\hat n}_1}\ \mid\ldots\mid\ D^{\hat n}_{\hat k_{\hat n}}\ \tilde\tau^{\hat n}_{\hat k_{\hat n},1}\ \ldots\ \tilde\tau^{\hat n}_{\hat k_{\hat n},\hat m^{\hat n}_{\hat k_{\hat n}}}\end{array}$$

It is important to note that the admissibility relation $\vdash$ is not defined within HOL, but as an extra-logical concept. Before attempting to construct a datatype, an ML function of the **datatype** package checks if the user input respects the rules described above. The point of this check is *not* to ensure correctness of the construction, but to provide high-level error messages.

Non-emptiness HOL does not admit empty types. Each of the new datatypes $(\alpha_1,\ldots,\alpha_h)t_j$ with $1 \leq j \leq n$ is guaranteed to be non-empty iff it has a constructor C^j_i with the following property: for all argument types $\tau^j_{i,i'}$ of the form $(\alpha_1,\ldots,\alpha_h)t_{j'}$ the datatype $(\alpha_1,\ldots,\alpha_h)t_{j'}$ is non-empty.

If there are no nested occurrences of the newly defined datatypes, obviously at least one of the newly defined datatypes $(\alpha_1,\ldots,\alpha_h)t_j$ must have a constructor

C_i^j without recursive arguments, a *base case*, to ensure that the new types are non-empty. If there are nested occurrences, a datatype can even be non-empty without having a base case itself. For example, with list being a non-empty datatype, **datatype** t = C (t list) is non-empty as well.

Just like ‟ described above, non-emptiness of datatypes is checked by an ML function before invoking the actual HOL **typedef** primitive, which would never accept empty types in the first place, but report a low-level error.

4.2 Limitations of Set-Theoretic Datatypes

Constructing datatypes in set-theory has some well-known limitations wrt. nesting of the *full* function space. This is reflected in the definition of admissible types given above. The last two cases of ‟ relate to *nested* (or *indirect*) occurrences of some of the newly defined types ($_1; \ldots ;$ $_h)t_{j'}$ in a type expression of the form $(\ldots ; \ldots$ ($_1; \ldots ;$ $_h)t_{j'}$ $\ldots ; \ldots)t^0$, where t^0 may either be the type constructor of an already existing datatype or the type constructor ! for the full function space. In the latter case, none of the newly defined types may occur in the first argument of the type constructor !, i.e. all occurrences must be *strictly positive*. If we were to drop this restriction, the datatype could not be constructed (cf. [6]). Recall that in classical set-theory

- there is *no* injection of type (t !) ,! t according to *Cantor's theorem*, if has more than one element;
- there *is* an injection in$_1$ :: (t !) ,! ((! t) !), because there is an injection (c x: c) :: t ,! (! t);
- there *is* an injection in$_2$ :: (t !) ,! ((t !) !), if has more than one element, since (x y: x = y) :: (t !) ,! ((t !) ! bool) is an injection and there is an injection bool ,! .

Thus datatypes with any constructors of the following form

datatype t = C (t ! bool) *j* D ((bool ! t) ! bool) *j* E ((t ! bool) ! bool)

cannot be constructed, because we would have injections C, D in$_1$ and E in$_2$ of type (t ! bool) ,! t, in contradiction to Cantor's theorem. In particular, inductive types in set-theory do *not* admit only weakly positive occurrences of nested function spaces. Moreover, nesting via datatypes exposes another subtle point when instantiating even *non-recursive* occurrences of function types: while **datatype** (;)t = C (! bool) *j* D (list) is legal, the specification of **datatype** γ u = E ((γ u;γ)t) *j* F is not, because it would yield the injection E C :: (γ u ! bool) ,! γ u; **datatype** γ u = E ((γ;γ u)t) *j* F is again legal.

Recall that our notion of admissible datatype specifications is an extra-logical one | reflecting the way nesting is handled in the construction (see x5.4). In contrast, [23] *internalizes* nested datatypes into the logic, with the unexpected effect that even non-recursive function spaces have to be excluded.

The choice of internalizing vs. externalizing occurs very often when designing logical systems. In fact, an important aspect of formal-logic engineering is to get the overall arrangement of concepts at *different layers* done right. The notions of *deep* vs. *shallow embedding* can be seen as a special case of this principle.

5 Constructing Datatypes in HOL

We now discuss the construction of the class of inductive types given in §4.1. According to §3.3, new types are defined in HOL “semantically” by exhibiting a suitable representing set. Starting with a universe that is closed wrt. certain injective operations, we cut out the representing sets of datatypes inductively using *Knaster-Tarski* (cf. [17]). Thus having obtained free inductive types, we construct several derived concepts, in particular primitive recursion.

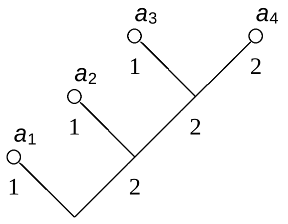
5.1 Universes for Representing Recursive Types

We describe the type $(\alpha, \beta)\mathsf{dtree}$ of trees, which is a variant of the universe formalized by Paulson [19], extended to support arbitrary branching. Type dtree provides the following operations:

$\mathsf{Leaf} :: \alpha \to (\alpha, \beta)\mathsf{dtree}$	embedding non-recursive occurrences of types
$\mathsf{In0}, \mathsf{In1} :: (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree}$	modeling distinct constructors
$\mathsf{Pair} :: (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree} \to (\alpha, \beta)\mathsf{dtree}$	modeling constructors with multiple arguments
$\mathsf{Lim} :: (\beta \to (\alpha, \beta)\mathsf{dtree}) \to (\alpha, \beta)\mathsf{dtree}$	embedding function types (infinitary products)

All operations are injective, e.g. $\mathsf{Pair}\ t_1\ t_2 = \mathsf{Pair}\ t_1'\ t_2' \iff t_1 = t_1' \wedge t_2 = t_2'$ and $\mathsf{Lim}\ f = \mathsf{Lim}\ f' \iff f = f'$. Furthermore, $\mathsf{In0}\ t \neq \mathsf{In1}\ t'$ for any t and t'.

Modeling Trees in HOL Set-theory A tree essentially is a set of *nodes*. Each node has a *value* and can be accessed via a unique *path*. A path can be modeled by a function that, given a certain *depth* index of type nat, returns a branching *label* (e.g. also nat). The figure below shows a finite tree and its representation.



$$T = \{(f_1, a_1), (f_2, a_2), (f_3, a_3), (f_4, a_4)\}$$

where

$$\begin{aligned} f_1 &= (1, 0, 0, \ldots) \\ f_2 &= (2, 1, 0, 0, \ldots) \\ f_3 &= (2, 2, 1, 0, 0, \ldots) \\ f_4 &= (2, 2, 2, 0, 0, \ldots) \end{aligned}$$

Here, a branching label of 0 indicates end-of-path. In the sequel, we will allow a node to have either a value of type bool or any type α. As branching labels, we will admit elements of type nat or any type β. Hence we define type abbreviations

$$\begin{aligned} (\alpha, \beta)\mathsf{node} &= (\mathsf{nat} \to (\beta + \mathsf{nat})) \times (\alpha + \mathsf{bool}) \\ (\alpha, \beta)\mathsf{dtree} &= ((\alpha, \beta)\mathsf{node})\mathsf{set} \end{aligned}$$

where the first component of a node represents the path and the second component represents its value. We can now define operations

push :: (+ nat) ! (;)node ! (;)node
push x n (i: (case i of 0) x j Suc j) fst n j); snd n)
Pair :: (;)dtree ! (;)dtree ! (;)dtree
Pair t_1 t_2 (push (Inr 1) \ t_1) [(push (Inr 2) \ t_2)

The function push adds a new head element to the path of a node, i.e.

push x ((y_0; y_1; : : :); a) = ((x; y_0; y_1; : : :); a)

The function Pair joins two trees t_1 and t_2 by adding the distinct elements 1 and 2 to the paths of all nodes in t_1 and t_2, respectively, and then forming the union of the resulting sets of nodes. Furthermore, we define functions Leaf and Tag for constructing atomic trees of depth 0:

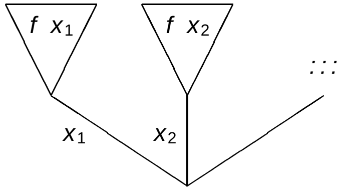
Leaf :: ! (;)dtree Tag :: bool ! (;)dtree
Leaf a f(x: Inr 0; Inl a)g Tag b f(x: Inr 0; Inr b)g

Basic set-theoretic reasoning shows that Pair, Leaf and Tag are indeed injective. We also define In0 and In1 which turn out to be injective and distinct.

In0 :: (;)dtree ! (;)dtree In1 :: (;)dtree ! (;)dtree
In0 t Pair (Tag false) t In1 t Pair (Tag true) t

Functions (i.e. infinitary products) are embedded via Lim as follows:

Lim :: (S ! (;)dtree) ! (;)dtree
Lim f fz j 9x: z = push (Inl x) \ (f x)g



That is, for all x the prefix x is added to the path of all nodes in f x, and the union of the resulting sets is formed.

Note that some elements of (;)dtree, such as trees with nodes of infinite depth, do not represent proper elements of datatypes. However, these junk elements are excluded when inductively defining the representing set of a datatype.

5.2 Constructing an Example Datatype

As a simple example, we will now describe the construction of the type list, specified by datatype list = Nil j Cons (list).

The Representing Set The datatype list will be represented by the set list-rep :: ((; unit)dtree)set. Since is the only type occurring non-recursively in the specification of list, the first argument of dtree is just . If more types would occur non-recursively, the first argument would be the sum of these types. Since there is no nested recursion involving function types, the second argument of dtree is just the dummy type unit. We define list-rep inductively:

$$\frac{}{\textsf{Nil-rep}\ 2\ \textsf{list-rep}} \qquad \frac{ys\ 2\ \textsf{list-rep}}{\textsf{Cons-rep}\ y\ ys\ 2\ \textsf{list-rep}} \qquad \begin{array}{l}\textsf{Nil-rep}\\ \textsf{Cons-rep}\ y\ ys\end{array} \qquad \begin{array}{l}\textsf{In0 dummy}\\ \textsf{In1 (Pair (Leaf}\ y)\ ys)\end{array}$$

Constructors Invoking the type definition mechanism described in x3.3 introduces the abstraction and representation functions

$$\begin{array}{lll}\textsf{Abs-list} & :: & (\ ;\textsf{unit})\textsf{dtree}\ !\quad \textsf{list}\\ \textsf{Rep-list} & :: & \quad\textsf{list}\ !\ (\ ;\textsf{unit})\textsf{dtree}\end{array}$$

as well as the axioms *Rep-list-inverse*, *Abs-list-inverse* and *Rep-list*. Using these functions, we can now define the constructors Nil and Cons:

$$\begin{array}{ll}\textsf{Nil} & \textsf{Abs-list Nil-rep}\\ \textsf{Cons}\ x\ xs & \textsf{Abs-list (Cons-rep}\ x\ \textsf{(Rep-list}\ xs))\end{array}$$

Freeness We can now prove that Nil and Cons are distinct and that Cons is injective, i.e. Nil 6= Cons x xs and Cons x xs = Cons x^0 xs^0 ( ) $x = x^0$ ^ $xs = xs^0$. Because of the isomorphism between list and list-rep, the former easily follows from the fact that In0 and In1 are distinct, while the latter is a consequence of the injectivity of In0, In1 and Pair.

Structural Induction For list an induction rule of the form

$$\frac{P\ \textsf{Nil}\quad 8x\ xs{:}\ P\ xs\ {=})\ P\ (\textsf{Cons}\ x\ xs)}{P\ xs}\ (\textit{list-ind})$$

can be proved using the induction rule

$$\frac{Q\ \textsf{Nil-rep}\quad 8y\ ys{:}\ Q\ ys\ \hat{}\ ys\ 2\ \textsf{list-rep}\ {=})\ Q\ (\textsf{Cons-rep}\ y\ ys)}{ys\ 2\ \textsf{list-rep}\ {=})\ Q\ ys}\ (\textit{list-rep-ind})$$

for the representing set list-rep derived by the inductive definition package from the rules described in x3.4. To prove *list-ind*, we show that P xs can be deduced from the assumptions P Nil and $8x$ xs: P xs =) P (Cons x xs) by the derivation

$$\dfrac{\dfrac{:::\quad \dfrac{\dfrac{:::\ {=})\ P\ (\textsf{Cons}\ y\ (\textsf{Abs-list}\ ys))}{:::\ {=})\ P\ (\textsf{Abs-list (Cons-rep}\ y\ (\textsf{Rep-list (Abs-list}\ ys))))}}{8y\ ys{:}\ P\ (\textsf{Abs-list}\ ys)\ \hat{}\ ys\ 2\ \textsf{list-rep}\ {=})\ P\ (\textsf{Abs-list (Cons-rep}\ y\ ys))}}{\textsf{Rep-list}\ xs\ 2\ \textsf{list-rep}\ {=})\ P\ (\textsf{Abs-list (Rep-list}\ xs))}}{P\ xs}$$

Starting with the goal P xs, we first use the axioms *Rep-list-inverse* and *Rep-list*, introducing the local assumption Rep-list xs 2 list-rep and unfolding xs to Abs-list (Rep-list xs). Now *list-rep-ind* can be applied, where Q and ys are instantiated with P Abs-list and Rep-list xs, respectively. This yields two new subgoals, one for the Nil-rep case and one for the Cons-rep case. We will only consider the Cons-rep case here: using axiom *Abs-list-inverse* together with the local assumption ys 2 list-rep, we unfold ys to Rep-list (Abs-list ys). Applying the definition of Cons, we fold Abs-list (Cons-rep y (Rep-list (Abs-list ys))) to

Cons y (Abs-list ys). Obviously, P (Cons y (Abs-list ys)) follows from the local assumption P (Abs-list ys) using the assumption $8x\ xs{:}\ P\ xs\ {=})\ \ P$ (Cons $x\ xs$).

In principle, inductive types are already fully determined by freeness and structural induction. Applications demand additional derived concepts, of course, such as case analysis, size functions, and primitive recursion.

Primitive Recursion A function on lists is *primitive recursive* iff it can be expressed by a suitable instantiation of the recursion combinator

list-rec :: ! (! list ! !) ! list !
list-rec $f_1\ f_2$ Nil = f_1
list-rec $f_1\ f_2$ (Cons $x\ xs$) = $f_2\ x\ xs$ (list-rec $f_1\ f_2\ xs$)

As has been pointed out in [8], a rather elegant way of constructing the function list-rec is to build up its graph list-rel by an inductive definition and then define list-rec in terms of list-rel using Hilbert's choice operator ":

$$\frac{}{(\mathrm{Nil};\ f_1)\ 2\ \text{list-rel}\ f_1\ f_2} \qquad \frac{(xs;\ y)\ 2\ \text{list-rel}\ f_1\ f_2}{(\mathrm{Cons}\ x\ xs;\ f_2\ x\ xs\ y)\ 2\ \text{list-rel}\ f_1\ f_2}$$

$$\text{list-rec}\ f_1\ f_2\ xs \qquad "y{:}\ (xs;\ y)\ 2\ \text{list-rel}\ f_1\ f_2$$

To derive the characteristic equations for list-rec given above, we show that list-rel does indeed represent a total function, i.e. for every list xs there is a unique y such that $(xs;\ y)\ 2$ list-rel $f_1\ f_2$. The proof is by structural induction on xs.

5.3 Mutual Recursion

Mutually recursive datatypes, such as aexp and bexp introduced in $x2$ are treated quite similarly as above. Their representing sets aexp-rep and bexp-rep of type ((;unit)dtree)set as well as the graphs aexp-rel and bexp-rel of the primitive recursion combinators are defined by *mutual* induction. For example, the rules for constructor Less are:

$$\frac{b_1\ 2\ \text{aexp-rep} \quad b_2\ 2\ \text{aexp-rep}}{\text{In0 (Pair}\ b_1\ b_2)\ 2\ \text{bexp-rep}} \qquad \frac{(x_1;\ y_1)\ 2\ \text{aexp-rel}\ f_1{:}{:}{:}f_6 \quad (x_2;\ y_2)\ 2\ \text{aexp-rel}\ f_1{:}{:}{:}f_6}{(\text{Less}\ x_1\ x_2;\ f_5\ x_1\ x_2\ y_1\ y_2)\ 2\ \text{bexp-rel}\ f_1{:}{:}{:}f_6}$$

5.4 Nested Recursion

Datatype (;)term is a typical example for nested (or indirect) recursion:

datatype (;)term = Var j App (((;)term)list)

As pointed out in [6, 7], datatype specifications with *nested* recursion can conceptually be unfolded to equivalent *mutual* datatype specifications without nesting. We also follow this extra-logical approach, avoiding the complications of internalized nesting [23]. Unfolding the above specification would yield:

datatype (;)term = Var j App ((;)term-list)
and (;)term-list = Nil^0 j Cons^0 ((;)term) ((;)term-list)

However, it would be a bad idea to actually introduce the type (;)term-list and the constructors Nil^0 and Cons^0, because this would prevent us from reusing common list lemmas in proofs about terms. Instead, we will prove that the representing set of (;)term-list is isomorphic to the type ((;)term)list.

The Representing Set We inductively define the sets term-rep and term-list-rep of type ((+ ; unit)dtree)set by the rules

$$\frac{}{\text{In0 (Leaf (Inl } a\text{))} \ 2 \ \text{term-rep}} \qquad \frac{ts \ 2 \ \text{term-list-rep}}{\text{In1 (Pair (Leaf (Inr } b\text{)) } ts\text{)} \ 2 \ \text{term-rep}}$$

$$\frac{}{\text{In0 dummy} \ 2 \ \text{term-list-rep}} \qquad \frac{t \ 2 \ \text{term-rep} \quad ts \ 2 \ \text{term-list-rep}}{\text{In1 (Pair } t \ ts\text{)} \ 2 \ \text{term-list-rep}}$$

Since there are two types occurring non-recursively in the datatype specification, namely and , the first argument of dtree becomes + .

Defining a Representation Function Invoking the type definition mechanism for term introduces the functions

```
Abs-term :: ( + ; unit)dtree ! ( ; )term
Rep-term :: ( ; )term ! ( + ; unit)dtree
```

for abstracting elements of term-rep and for obtaining the representation of elements of (;)term. To get the representation of a list of terms we now define

```
Rep-term-list :: (( ; )term)list ! ( + ; unit)dtree
Rep-term-list Nil = In0 dummy
Rep-term-list (Cons t ts) = In1 (Pair (Rep-term t) (Rep-term-list ts))
```

Determining the representation of Nil is trivial. To get the representation of Cons *t ts*, we need the representations of *t* and *ts*. The former can be obtained using the function Rep-term introduced above, while the latter is obtained by a recursive call of Rep-term-list. Obviously, Rep-term-list is primitive recursive and can therefore be defined using the combinator list-rec:

```
Rep-term-list     list-rec (In0 dummy) ( t ts y: In1 (Pair (Rep-term t) y))
Abs-term-list     inv Rep-term-list
```

It is a key observation that Abs-term-list and Rep-term-list have the properties

```
Abs-term-list (Rep-term-list xs) = xs
ys 2 term-list-rep =) Rep-term-list (Abs-term-list ys) = ys
Rep-term-list xs 2 term-list-rep
```

i.e. ((;)term)list and term-list-rep are isomorphic, which can be proved by structural induction on list and by induction on rep-term-list. Looking at the HOL type definition mechanism once again (x3.3), we notice that these properties have exactly the same form as the axioms which are introduced for actual newly defined types. Therefore, all of the following proofs are the same as in the case of mutual recursion without nesting, which simplifies matters considerably.

Constructors Finally, we can define the constructors for term:

```
Var a        Abs-term (In0 (Leaf (Inl a)))
App b ts     Abs-term (In1 (Pair (Leaf (Inr b)) (Rep-term-list ts)))
```

5.5 Infinitely Branching Types

We show how to construct infinitely branching types such as (; ; y)tree, cf. x2.

The Representing Set tree-rep will be of type ((+ ; y)dtree)set. Since the two types and occur non-recursively in the specification, the first argument of dtree is the sum + of these types. The only branching type, i.e. a type occurring on the left of some ! , is y. Therefore, y is the second argument of dtree. We define tree-rep inductively by the rules

$$\frac{}{\text{In0 (Leaf (Inl } a)) \ 2 \text{ tree-rep}} \qquad \frac{g\ 2 \text{ Funs tree-rep}}{\text{In1 (Pair (Leaf (Inr } b)) \text{ (Lim } g)) \ 2 \text{ tree-rep}}$$

where the premise g 2 Funs tree-rep means that all function values of g represent trees. The *monotone* function Funs is defined by

Funs :: set ! (!)set
Funs S fg j range g Sg

Constructors We define the constructors of tree by

Atom a Abs-tree (In0 (Leaf (Inl a)))
Branch b f Abs-tree (In1 (Pair (Leaf (Inr b)) (Lim (Rep-tree f))))

The definition of Atom is straightforward. To form a Branch from element b and subtrees denoted by f, we first determine the representation of the subtrees by composing f with Rep-tree and then represent the resulting function using Lim.

Structural Induction The induction rule for type tree shown in x2 can be derived from the corresponding induction rule for the representing set tree-rep by instantiating Q and u with P Abs-tree and Rep-tree, respectively:

$$\frac{8a:\ Q\ \text{(In0 (Leaf (Inl } a))) \qquad 8b\ g:\ g\ 2 \text{ Funs (tree-rep} \setminus \ fx\ j\ Q\ xg) =)\ Q\ \text{(In1 (Pair (Leaf (Inr } b)) \text{ (Lim } g)))}{u\ 2 \text{ tree-rep} =)\ Q\ u}$$

The unfold/fold proof technique already seen in x5.2 can also be extended to functions: if g 2 Funs tree-rep, then g = Rep-tree (Abs-tree g).

Primitive Recursion Again, we define the tree-rec combinator given in x2 by constructing its graph tree-rel inductively:

$$\frac{}{\text{(Atom } a;\ f_1\ a)\ 2 \text{ tree-rel } f_1\ f_2} \qquad \frac{f^0\ 2 \text{ compose } f \text{ (tree-rel } f_1\ f_2)}{\text{(Branch } b\ f;\ f_2\ b\ f\ f^0)\ 2 \text{ tree-rel } f_1\ f_2}$$

The *monotone* function compose used in the second rule is defined by

compose :: (!) ! (y)set ! (! y)set
compose f R ff⁰ j 8x: (f x; f⁰ x) 2 Rg

The set compose f R consists of all functions that can be obtained by composing the function f with the relation R. Since R may not necessarily represent a total function, compose f R can also be empty or contain more than one function.

However, if for every x there is a unique y such that $(f\ x, y) \in$ tree-rel $f_1\ f_2$, then there is a unique f' with $f' \in$ compose f (tree-rel $f_1\ f_2$). This is the key property used to prove that tree-rel $f_1\ f_2$ represents a total function.

Even More Complex Function Types The construction described above can be made slightly more general. Assume we want to define the datatype t, whose **datatype** specification contains function types $\sigma^i_1 \to \cdots \to \sigma^i_{m_i} \to$ t, where $1 \le i \le n$. The representing set t-rep then has the type

$$((\ldots, (\sigma^1_1 \times \cdots \times \sigma^1_{m_1}) + \cdots + (\sigma^n_1 \times \cdots \times \sigma^n_{m_n}))\mathrm{dtree})\mathrm{set}$$

The representation of a function $f_i :: \sigma^i_1 \to \cdots \to \sigma^i_{m_i} \to$ t is

$$\mathrm{Lim}\ (\text{sum-case}\ \underbrace{\text{dummy} \ldots \text{dummy}}_{i-1\text{ times}}\ (\text{Rep-t} \circ ((\underbrace{\text{uncurry} \circ \cdots \circ \text{uncurry}}_{m_i-1\text{ times}})\ f_i))\ \underbrace{\text{dummy} \ldots \text{dummy}}_{n-i\text{ times}})$$

where

$$\begin{array}{lll} \text{uncurry} & :: & (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma \\ \text{sum-case} & :: & (\alpha_1 \to \beta) \to \cdots \to (\alpha_n \to \beta) \to \alpha_1 + \cdots + \alpha_n \to \beta \end{array}$$

are injective, i.e. uncurry f = uncurry g iff $f = g$, and sum-case $f_1 \ldots f_n$ = sum-case $g_1 \ldots g_n$ iff $f_1 = g_1 \wedge \ldots \wedge f_n = g_n$.

6 Building a Working Environment

6.1 Inverting Datatype Definitions

The hierarchy of definitional packages, as illustrated in §1, and the dependency of auxiliary theories used for the construction are often in conflict. For example, we have used basic types such as nat, $\alpha + \beta$, $\alpha \times \beta$ for the universe underlying datatypes. Any of these could be characterized as inductive types, but had to be built “manually”, because **datatype** had not yet been available at that stage.

Of course, we would like to keep the Isabelle/HOL standard library free of any such accidental arrangements due to *bootstrap problems* of the HOL logic. Note that with most types being actual datatypes — offering the standard repertoire of derived concepts such as induction, recursion, pattern matching by cases etc. — the resulting system would become conceptually much simpler, with less special cases. Furthermore, proper datatypes may again partake in indirect recursion. Users certainly expect to be able to nest types via $\alpha + \beta$ and $\alpha \times \beta$.

We propose *inverted datatype definitions* as an answer to the above problem. Given a type together with freeness and induction theorems, the **rep-datatype** mechanism figures out the set of constructors and does all the rest of standard datatype constructions automatically. Thus we avoid cycles in theory/package dependencies in a clean way. Note that the same mechanism is used internally when unwinding indirect recursion (reconsider term-list vs. term list in §5.4).

From a purely logical point of view one would probably approach bootstrap issues differently. For example, [8] provides a very careful development of the theory underlying the datatype construction, reducing the requirements to a bare minimum, even avoiding natural numbers. Interestingly the actual implementation does not fully follow this scheme, but *does* use natural numbers.

6.2 Cooperation of Definitional Packages

As depicted in *x*1, some Isabelle/HOL packages are built on top of each other. For example, `record` [13] constructs extensible records by defining a separate (non-recursive) datatype for any record field. Other packages such as `recdef` [21, 22] refer to certain information about datatypes which are involved in well-founded recursion (e.g. size functions). We see that some provisions have to be made in order to support *cooperation* of definitional packages properly.

In particular, there should be means to store auxiliary information in theories. Then packages such as `datatype` would associate sufficient source level information with any type, such as the set of constructors, induction rule, and primrec combinator. Thus we get a more robust and scalable system than by trying to weed through the primitive logical declarations emitted by the package. Isabelle98-1 does already support an appropriate \theory data" concept.[2]

With extra-logical information associated with logical objects, we may also offer users a more uniform view to certain general principles. For example, \proof by induction" or \case analysis" may be applied to some x, with the actual tactic figured out internally. Also note that by deriving definitional mechanisms from others, such as `record` from `datatype`, these operations are *inherited*. Thus case analysis etc. on record fields would become the same as on plain datatypes.

7 Conclusion and Related Work

We have discussed Isabelle/HOL's new definitional packages for inductive types and (primitive) recursive functions (see also [2]) at two different levels of concept.

At the logical level, we have reviewed a set-theoretic construction of mutual, nested, arbitrarily branching types together with primitive recursion combinators. Starting with a schematic universe of trees | similar to [19], but extended to support infinitely branching | we have cut out representing sets for inductive types using the usual Knaster-Tarski fixed-point approach [17, 8].

Stepping back from the pure logic a bit, we have also discussed further issues we considered important to achieve a scalable and robust working environment. While this certainly does not yet constitute a systematic discipline of \Formal-Logic Engineering", we argue that it is an important line to follow in order to provide theorem proving systems that are \successful" at a larger scale. With a slightly different focus, [1] discusses approaches to \Proof Engineering".

The importance of advanced definitional mechanisms for applications has already been observed many years ago. Melham [12] pioneers a HOL datatype

[2] Interestingly, while admitting arbitrary ML values to be stored, this mechanism can be made *type-safe* within ML (see also [11]).

package (without nesting), extended later by Gunter [6, 7] to more general branching. Paulson [17] establishes Knaster-Tarski as the primary principle underlying (co)inductive types; the implementation in Isabelle/ZF set-theory also supports infinite branching. Völker [23] proposes a version of datatypes for Isabelle/HOL with nested recursion *internalized* into the logic, resulting in some unexpected restrictions of *non-recursive* occurrences of function spaces. Harrison [8] undertakes a very careful logical development of mutual datatypes based on cardinality reasoning, aiming to reduce the auxiliary theory requirements to a minimum. The implementation (HOL Light) has recently acquired nesting, too.

Our Isabelle/HOL packages for datatype and primrec have been carefully designed to support a superset of functionality, both with respect to the purely logical virtues and as its integration into a scalable system. This is intended not as the end of the story, but the beginning of the next stage.

Codatatypes would follow from Knaster-Tarski by duality quite naturally (e.g. [17]), as long as simple cases are considered. Nesting codatatypes, or even mixing datatypes and codatatypes in a useful way is very difficult. While [10] proposes a way of doing this, it is unclear how the informal categorical reasoning is to be transferred into the formal set-theory of HOL (or even ZF).

Non-freely generated types would indeed be very useful if made available for nesting. Typical applications refer to some type that contains a finitary environment of itself. Currently this is usually approximated by nesting ( ) list.

Actual *combination of definitional* packages would be another important step towards more sophisticated standards, as are established in functional language compiler technology, for example. While we have already achieved decent cooperation of packages that are built on top of each other, there is still a significant gap towards arbitrary combination (mutual and nested use) of separate packages. In current Haskell compilers, for example, any module (read “theory”) may consist of arbitrary declarations of classes, types, functions etc. all of which may be referred to mutually recursive. Obviously, theorem prover technology will still need some time to reach that level, taking into account that “compilation” means actual theorem proving work to be provided by the definitional packages.

References

[1] H. P. Barendregt. The quest for correctness. In *Images of SMC Research*, pages 39–58. Stichting Mathematisch Centrum, Amsterdam, 1996.

[2] S. Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL (in German). Master's thesis, Technische Universität München, 1998.

[3] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[4] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, and C. Muñoz. *The Coq Proof Assistant User's Guide, version 6.1.* INRIA-Rocquencourt et CNRS-ENS Lyon, 1996.

[5] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[6] E. L. Gunter. Why we can't have SML style datatype declarations in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume A-20 of *IFIP Transactions*, pages 561{568. North-Holland Press, 1992.

[7] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *LNCS*, pages 141{154. Springer, 1994.

[8] J. Harrison. Inductive definitions: automation and application. In P. J. Windley, T. Schubert, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *LNCS*, pages 200{213, Aspen Grove, Utah, 1995. Springer.

[9] J. Harrison. HOL done right. Unpublished draft, 1996.

[10] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, volume 1290 of *LNCS*, pages 220{241. Springer, 1997.

[11] F. Kammüller and M. Wenzel. Locales | a sectioning concept for Isabelle. Technical Report 449, University of Cambridge, Computer Laboratory, October 1998.

[12] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341{386. Springer, 1989.

[13] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*. Springer, 1998.

[14] T. Nipkow and D. von Oheimb. Java$_{'ight}$ is type-safe | definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, New York, 1998.

[15] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[16] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9R, Computer Science Laboratory, SRI International, 1993.

[17] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 148{161, Nancy, France, 1994. Springer.

[18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[19] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175{204, 1997.

[20] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*. Springer, 1990.

[21] K. Slind. Function definition in higher order logic. In J. Wright, J. Grundy, and J. Harrison, editors, *9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'96*, volume 1125 of *LNCS*. Springer, 1996.

[22] K. Slind. Derivation and use of induction schemes in higher-order logic. In *10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'97*, volume 1275 of *LNCS*. Springer, 1997.

[23] N. Völker. On the representation of datatypes in Isabelle/HOL. In L. C. Paulson, editor, *First Isabelle Users Workshop*, 1995.

[24] M. Wenzel. Type classes and overloading in higher-order logic. In *10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'97*, volume 1275 of *LNCS*. Springer, 1997.

Isomorphisms –
A Link Between the Shallow and the Deep

Thomas Santen

Softwaretechnik, FR 5-6
Fachbereich Informatik
Technische Universität Berlin
Franklinstraße 28/29, D-10587 Berlin, Germany
santen@cs.tu-berlin.de

Abstract. We present a theory of isomorphisms between typed sets in Isabelle/HOL. Those isomorphisms can serve to link a shallow embedding with a theory that defines certain concepts directly in HOL. Thus, it becomes possible to use the advantage of a shallow embedding that it allows for efficient proofs about concrete terms of the embedded formalism with the advantage of a deeper theory that establishes general abstract propositions about the key concepts of the embedded formalism as theorems in HOL.

1 Introduction

A strong and decidable type system is one of the distinguishing features of the versions of higher-order logic that are implemented in systems such as HOL [4] and Isabelle/HOL [15]. The implicit propositions that types represent keep the concrete syntax of terms simple. Decidable type-checking is an efficient means of finding errors early when developing a theory. Proofs in those logics tend to be shorter and are more easily automated than the corresponding ones in untyped theories – at least at the current state of the art [8].

However, types become an obstacle when one wishes to capture concepts formally whose definitions inherently need to abstract from types. The kind of polymorphism that HOL[1] provides to abstract from types often is too weak to express the independence of those concepts from specific types in its full generality. Apparently general theories turn out to be less generally applicable than one might naively expect, because the definitions of their foundational concepts are “only” polymorphic. For example, theories of concurrent processes [10, 20] use definitions that are polymorphic in the type of data (channels, actions) that processes exchange. Operations on several processes, such as their parallel composition, then require the involved processes to work on the same type of data. In theorems about concepts such as refinement, type unification often forces the involved processes to have the same type, because a polymorphic variable must have the same type at all of its occurrences in a theorem. Consequently, (straight-forward) applications of those theories are restricted to systems that exchange a single type of

[1] From now on, we use HOL as a synonym for the logic, not the HOL system.

Y. Bertot et al. (Eds.): TPHOLs’99, LNCS 1690, pp. 37–54, 1999.

data. This restriction is not a fault of the designers of those theories but an immediate consequence of the type system of HOL.

There is no clear-cut borderline between shallow and deep embeddings. In the following, we discuss some aspects of that distinction that are important in the context of the present paper (see [1] for another view on “shallow vs. deep”).

A *shallow embedding* of a typed language in HOL identifies the types of the embedded language with types of HOL. Such an embedding has the advantage that proving theorems about particular sentences of the language directly corresponds to a proof about the corresponding HOL term and is thus as efficient as proving any other proposition in HOL. A shallow embedding does not restrict the types in applications in the way the “deeper” theories do that we mentioned before. At a closer look, the reason for that “advantage” turns into a major drawback of shallow embeddings: they do not provide definitions of the foundational concepts of the embedded language at all. Therefore, it usually is impossible reason about those concepts in general. Facts about them must be re-proven for each particular instance. If the proofs of these facts are involved, re-proving them over and over again may considerably diminish the advantage of efficiency in practical applications that a shallow embedding promises. For example, shallow embeddings of object-oriented concepts [5, 12] in higher-order logic do not define the concepts of a class or an object in general. LOOP [5], an embedding of co-algebraic specifications in PVS [14], generates definitions and proof scripts for each particular class specification that introduce general concepts, such as bisimilarity, and derive properties about them that hold for any class. The more complex the theory encoded in those proof scripts is, the more effort is needed to represent a particular class specification: the effort depends not only on the size of the class but also on the complexity of the general theory of classes.

A *deep embedding*, in contrast, provides definitions of the key concepts of the embedded formalisms in HOL. In the example, a deep embedding defines the type of a class and the set of objects induced by a class in HOL. More complex definitions, such as class refinement, can be based on the definitions of classes. Furthermore, a HOL-theory about those concepts can be developed that establishes theorems in the logic that a shallow embedding, such as LOOP, can use meta-logically, in the form of tactics, only. Unfortunately, as mentioned before, the theorems of a deeper embedding usually are not directly applicable to concrete entities (e.g. classes) that are represented by a shallow embedding.

A very general solution to this dilemma and to other problems that arise from the type discipline of HOL would combine higher-order logic and an axiomatic set theory such as ZF in such a way that the set theory could be used to reason about concepts that do not fit well with the type system of HOL [3]. In the present paper, we propose a specific solution within HOL that addresses the combination of deep and shallow embeddings to obtain the best of both worlds in applications. We use a type of isomorphisms between sets, which we define in Isabelle/HOL, to formally capture the idea that definitions in HOL, such as the ones referred to before, restrict the general intuitive concepts behind them “without loss of generality” to specializations that polymorphism can express. Concealing type differences of entities that a shallow embedding produces, e.g. by constructing the sum of all involved types, and substituting isomorphisms for equalities makes a general theory developed in HOL applicable to specific problems that are represented via the shallow embedding.

In Sect. 2, we introduce our notation of Isabelle/HOL terms, and briefly recapitulate the type definition mechanism as it is implemented in Isabelle/HOL. In Sect. 3, we further motivate the use of isomorphisms to link a shallow embedding with a more general theory by the example of combining a HOL-theory of object-oriented specification with a shallow embedding of the specification language Z. In Sect. 4, we present the basic idea of modeling bijections between sets in HOL. Sect. 5 introduces partial equivalence relations and quotients as a means to describe the type of isomorphisms in Sect. 6, where we also define the basic operations on isomorphisms. In Sect. 7, we define functors that map isomorphisms to isomorphisms, in particular by "lifting" them to type constructors. Section 8 applies those functors to lift isomorphisms to predicates. We need this construction in Sect. 9 to precisely characterize isomorphic representations of "the same" class specification in the example of Sect. 3. We sketch how those isomorphisms allow us to derive properties of class refinement as HOL-theorems and to apply those theorems to concrete class specifications whose components are specified in Z. Pointing out more general applications of the theory of isomorphisms, we conclude in Sect. 10.

2 Notation

In our notation of HOL terms as defined in Isabelle/HOL, we use the standard graphical symbols. For two types and , is the Cartesian product type, and ! is the type of (total) functions with domain and range .

We denote a constant definition by a declaration c :: and a defining equation $c \overset{\text{def}}{=} t$. The symbol $\overset{\text{def}}{=}$ indicates that the defining equation is checked by Isabelle to preserve consistency of the theory.

A *type definition* introduces a new type constant that is isomorphic to a set B of elements of some other type. An axiom asserting the existence of a bijection between B and the elements of the new type establishes that isomorphism. In addition to some technical conditions concerning polymorphism, we have to prove that B is non-empty to be sure that the type definition preserves the existence of standard models. In Isabelle/HOL, the new type is defined by stating that it is isomorphic to a non-empty set B of elements of some (already defined) type . Two functions ¯ :: ! , the *abstraction* function, and _ :: ! , the *representation* function, describe the isomorphism. They are characterized by the axioms

$$(\_\, a)\ 2\ B \tag{1}$$

$$(^{-}(\_\, a)) = a \tag{2}$$

$$b\ 2\ B\ \ )\ \ (\_(^{-}\, b)) = b \tag{3}$$

The representation function maps the elements of to members of the set B. On the set B, abstraction and representation functions are inverses of each other. The condition $b\ 2\ B$ of (3) accounts for the totality of ¯.

We use the following notation for a type definition:

$$typ \overset{\text{typ}}{=} B \qquad \text{justified by } \mathsf{witness\_thm}$$

It declares the new type *typ*, the abstraction function $\overline{typ}$ :: ! *typ* and the representation function $\underline{typ}$ :: *typ* ! . It introduces three axioms corresponding to (1), (2), and (3). Processing the type definition, Isabelle proves that B is non-empty, using the derived theorem witness_thm in the process.

In contrast to a type definition, a *type synonym* does not introduce a new type but just a syntactic abbreviation of a type expression. The type equation $\overset{\text{syn}}{=}$ defines the type constructor to be equal to the type , which contains the type variable .

3 Example: Class Specifications

We illustrate the use of isomorphisms to link a shallow embedding with an abstract theory by the example of a theory of specifications of object-oriented systems. In earlier work [16], we presented a theory of model-based specifications of object-oriented systems that relies on the shallow embedding *HOL-Z* [7] of Z [19] in Isabelle/HOL for specifying the components of a class. The concept of a class (and of the set of objects induced by a class) are defined in HOL. This allows us to build an abstract theory of properties of classes, such as behavioral conformance,[2] and apply the theorems of that theory to concrete classes whose components are specified in *HOL-Z*. To show its practical applicability, we applied an extended version of that theory [18] to analyze the conformance relationships in the Eiffel data structure libraries [9]. In the following, we very briefly sketch the basic idea of linking *HOL-Z* with the abstract theory of object-oriented concepts. The specifics of that theory (and of the shallow embedding of Z) are irrelevant for the present paper. We refer the curious reader to [16, 18] for more details.

The type (; ; ; ; !) *Class* describes class specifications over a type of method identifiers , of constants , of (mutable) states , of inputs , and of outputs !. The state of an object has a constant and a mutable part. Thus, it is of the type .

$$
\begin{array}{lll}
(\;;\;;\;;\;;!)\,\mathit{Class} \overset{\text{typ}}{=} \{(C;S;I;M;H)\,j & & \\
 & (C ::\;\; !\; bool) & (S :: [\;;\;]\; !\; bool)\\
 & (I :: [\;;\;]\; !\; bool) & (M :: (\;;\;;\;;\;;!)\,\mathit{Methods})\\
 & (H :: [\;;\;;\;]\; !\; bool) :\; \mathit{Cls}\;\; C\;S\;I\;M\;H\} &
\end{array}
$$

A class specification consists of five components: predicates describing the admissible constants (C) and mutable states (S), the initialization condition for object creation (I), a history constraint (H), which is a relation on states, and a method suite (M) that maps method identifiers to operation specifications. According to the specification paradigm of Z, an operation specification is a relation on pre-states, inputs, post-states, and outputs. Relating the components of a class specification, the predicate *Cls* ensures that they conform to the intuition of “describing a class”. For our purpose, it is important to note that the components are predicates, i.e. functions to the boolean values, and that the typing rules of HOL force all operation specifications of a method suite M to have the same types of input and output.

Because it is a shallow embedding, *HOL-Z* maps the types of Z to types of HOL. This is justified, because the type systems of the two languages are very similar [17]. In

[2] Behavioral conformance is a variant of data refinement [6] that takes the specifics of object-oriented systems into account.

HOL-Z, the types of different operation specifications that describe the methods of a class, in general, differ in the types of inputs and outputs. The operation $Cls \boxplus (id; op)$ adds an operation specification op as a method called id to a class specification Cls. Its declaration shows the key idea of linking the shallow embedding with the abstract theory of classes:

$$\_ \boxplus \_ :: [(\ ; \ ; \ ; \ ; !)\,\mathit{Class}; \quad (\ ; \ ; {}^0; !^0)\,\mathit{Op}]$$
$$! \ (\ ; \ ; \ ; {}^0 + \ ; !^0 + !)\,\mathit{Class}$$

Forming the sums ${}^0 +$ and $!^0 + !$ of the input and output types of Cls and op, $Cls \boxplus (id; op)$ produces a class specification that works on larger types of inputs and outputs than Cls. Its definition transforms the method suite of Cls and the operation specification op such that their input and output parameters are injected into the appropriate summand of those sum types.

This definition captures the idea of much theoretical work that “without loss of generality” one can restrict an investigation to a single-sorted case, because it is always possible to reduce the many-sorted case to the single-sorted one by combining the relevant sorts in a disjoint sum. For the practical work with a theory in a theorem prover, however, it is important to link the abstract theory and the shallow embedding in a way that conveniently handles the many different types arising. For example, adding two operations in a different order to a class specification, e.g. $Cls \boxplus (id_1; op_1) \boxplus (id_2; op_2)$ and $Cls \boxplus (id_2; op_2) \boxplus (id_1; op_1)$, yields specifications of “the same” class that are not even comparable in HOL, because they have different types. Furthermore, relations, such as behavioral conformance, must be defined for classes of *different* types. At first sight, this makes it impossible to profit from an abstract calculus of relations, such as the square calculus of [13], when reasoning about classes.

The theory of isomorphisms that we present in this paper allows us to formally characterize that two class specifications specify “the same” class, and it allows us to map the classes of a (finite) system to a single type, thus making general concepts applicable that require identical types of the involved classes.

4 Bijections

We wish to characterize bijections between two sets $A ::$ set and $B ::$ set whose elements are of different types and . In axiomatic set theory, the situation is clear: an injective function with domain A and range B is a bijection between A and B. If such a function exists, then A and B are isomorphic sets.

In HOL, all functions are total. Thus, given a function $f ::$ $!$, there is no point in requiring that the “domain” of f be the set A. Requiring that f is injective would be too strong, because this would imply that the cardinality of the type must not be greater than the cardinality of the type . We are just interested in the relationship between the sets A and B, not their underlying types. Therefore, *card* A *card* , but not *card* *card* is a necessary condition for a bijection between A and B to exist. For A and B to be isomorphic, it suffices to require that f is injective *on the set* A, i.e. that the images of two distinct elements of A under f are distinct, and that the image of A under f is equal to B, $f(jAj) = B$. The predicate inj_onto captures injectivity on a set formally.

$$inj\_onto :: [\alpha \rightarrow \beta, \alpha\ set] \rightarrow bool$$

$$inj\_onto\ f\ A \overset{\mathrm{def}}{=} \forall x \in A.\ \forall y \in A.\ f\ x = f\ y \Rightarrow x = y$$

We are not just interested in the fact that A and B have the same cardinality, but we also wish to convert the elements of one set to the corresponding elements of the other. Therefore, we need to know the bijection f and the co-images of the elements of B under f. The function Inv maps a function to one of its inverses using Hilbert's choice operator: $Inv\ f \overset{\mathrm{def}}{=} (\lambda y.\ \varepsilon x.\ f\ x = y)$.

Because we cannot require that f is injective on the entire type α, and because B may encompass all elements of β, i.e. $B = UNIV$, the function f, in general, maps several elements of α to the same element of B. Let, for example, $f\ a = b$ and $f\ x = b$. For $b \in B$, it is not necessarily true that $Inv\ f\ b \in A$, because there are models in which $(\varepsilon z. f\ z = b) = x$ for $x \notin A$. The function inv_on chooses an inverse that maps elements into a given set whenever possible.

$$inv\_on :: [\alpha\ set; \alpha \rightarrow \beta] \rightarrow \beta \rightarrow \alpha$$

$$inv\_on\ A\ f \overset{\mathrm{def}}{=} (\lambda y.\ (\varepsilon x.\ f\ x = y \wedge (y \in f(|A|) \Rightarrow x \in A)))$$

In general, there are many bijections of isomorphic sets A and B that differ on the complement of A only. The relation $eq\_on\ A$ characterizes the functions that are equal on A. It abstracts from differences on the complement of A.

$$eq\_on :: [\alpha\ set; \alpha \rightarrow \beta; \alpha \rightarrow \beta] \rightarrow bool$$

$$eq\_on\ A\ f\ g \overset{\mathrm{def}}{=} \forall x \in A.\ f\ x = g\ x$$

With these preliminaries, we can characterize an *isomorphism* of the sets A and B by the set of all functions that are injective onto A, whose image of A is equal to B, and which are equal on A. This characterization is the basis to define a type of isomorphisms in Section 6.

5 Partial Equivalences, Quotients, and Congruences

We base our definition of isomorphisms on a theory of relations on sets that introduces equivalence relations, quotients, and congruences. This theory is a slight modification of a standard Isabelle/HOL theory that has been adapted from Isabelle/ZF. Our modification takes congruences with respect to polymorphic binary relations into account.

A relation r is an *equivalence* relation with domain A, $equiv\ A\ r$, if it is reflexive on A, symmetric, and transitive. A *quotient* A/r of a set A and a relation r is the set of relational images of the singletons of $\mathbb{P}\ A$.

$$equiv :: [\alpha\ set; (\alpha \times \alpha)\ set] \rightarrow bool \qquad \_/\_ :: [\alpha\ set; (\alpha \times \alpha)\ set] \rightarrow \alpha\ set\ set$$

$$equiv\ A\ r \overset{\mathrm{def}}{=} refl\ A\ r \wedge sym\ r \wedge trans\ r \qquad A/r \overset{\mathrm{def}}{=} \bigcup_{x \in A} \{r(|\{x\}|)\}$$

Although the relation r in the quotient usually is an equivalence relation, it is not necessary to require that property in the definition. For an equivalence relation r, the set $r(jfagj)$ denotes the equivalence class of a with respect to r. Note that we use the same notation for the image of a set A under a function f, $f(jAj)$, and the image of A under a relation $r :: (\quad)\, set$, $r(jAj)$. The following two facts allow us to reduce a proposition about the member sets of a quotient to representatives of equivalence classes. First, two equivalence classes of a relation r are equal if and only if two representatives are in the relation r.

$$\frac{equiv\ A\ r \qquad x\ 2\ A \qquad y\ 2\ A}{(r(jfxgj) = r(jfygj)) = ((x;y)\ 2\ r)} \tag{4}$$

Second, the class of $x\ 2\ A$ with respect to a relation r is an element the quotient $A{=}r$.

$$\frac{x\ 2\ A}{r(jfxgj)\ 2\ A{=}r} \tag{5}$$

If r is an equivalence class, then an operation on the quotient $A{=}r$ is usually defined by a corresponding operation on representatives of equivalence classes of r. Such a definition is well-formed if it is independent of the particular representatives of equivalence classes it refers to. Then r is a congruence relation with respect to the operation. The following predicates characterize congruences with respect to unary and binary operations.

$$\begin{aligned} congruent &:: [(\quad)\, set;\quad !\quad]\ !\ bool \\ congruent\ r\ b &\overset{\text{def}}{=} 8y\, z{:}\,(y;z)\ 2\ r\ \ )\ \ b\, y = b\, z \end{aligned} \tag{6}$$

$$\begin{aligned} congruent_2 &:: [(\quad)\, set; (\quad)\, set; [\ ;\ ]\ !\quad]\ !\ bool \\ congruent_2\ r\ r^0\ b &\overset{\text{def}}{=} 8y_1\, z_1\, y_2\, z_2{:}\,(y_1;z_1)\ 2\ r\ \ )\ \ (y_2;z_2)\ 2\ r^0 \\ &\qquad)\ \ b\, y_1\, y_2 = b\, z_1\, z_2 \end{aligned} \tag{7}$$

The direct transcription of the ZF definition of congruences with respect to binary operations uses only one parameter relation:

$$\begin{aligned} congruent_2^m &:: [(\quad)\, set; [\ ;\ ]\ !\quad]\ !\ bool \\ congruent_2^m\ r\ b &\overset{\text{def}}{=} 8y_1\, z_1\, y_2\, z_2{:}\,(y_1;z_1)\ 2\ r\ \ )\ \ (y_2;z_2)\ 2\ r \\ &\qquad)\ \ b\, y_1\, y_2 = b\, z_1\, z_2 \end{aligned} \tag{8}$$

This definition requires that both parameters of b are of the same type . The more liberal definition (7) of $congruent_2$ allows the parameters of b to be of different types. To achieve this, $congruent_2$ needs two parameter relations of different types. Using only one relation as in the defining axiom of (8) would force the parameter types of b to be identical by type unification.

In practice, both parameter relations r and r^0 of $congruent_2$ will be different polymorphic instances of the same (polymorphically defined) relation. The seemingly unnecessary duplication of the parameter relations is a consequence of the weak polymorphism of HOL. The type variables of an axiom are constants for the type inference in the axiom, because they do not denote universal quantifications over types [2]. At some places where a naive intuition would expect one parameter to suffice, we are forced to supply as parameters several structurally equal terms that differ only in their types.

Congruences show that the definitions of operations on equivalence classes are independent of the choice of representatives. The following fact captures this property formally for a common construction. If r is a congruence with respect to a set-valued operation b, then the union of the images of b applied to the elements of an equivalence class is just the image of a representative of that class.

$$\frac{equiv\ A\ r \qquad congruent\ r\ b \qquad a\ 2\ A}{\left(\bigcup_{x\,2\,r\,(j\,f\,a\,g\,j)} b\ x\right) = b\ a} \tag{9}$$

Similar statements hold for the other constructions that we use to define operations on isomorphisms in the following section.

6 The Type of Isomorphisms

The type $(\ ;\ )\,iso$ is the type of isomorphisms between sets of type $\ set$ and sets of type $\ set$. The characterizing set of $(\ ;\ )\,iso$ is a quotient: an isomorphism is an equivalence class of pairs of functions and domains. For each pair $(f;A)$ in a class, f is injective onto A. Two pairs are equivalent if the domains are equal and the functions are equal on the domains.

$$(\ ;\ )\,isopair \overset{\text{syn}}{=} (\ \ !\ \ )\ \ (\ \ set) \tag{10}$$

$$_{iso}\ ::\ ((\ ;\ )\,isopair \quad (\ ;\ )\,isopair)\,set$$

$$_{iso} \overset{\text{def}}{=} fp\ j\ p\ f\ g\ A\ B{:}\,p = ((f;A);(g;B)) \wedge A = B \wedge \\ inj\_onto\ f\ A \wedge eq\_on\ A\ f\ gg \tag{11}$$

$$(\ ;\ )\,iso \overset{\text{typ}}{=} f(f;A)\ j\ f\ A{:}\ inj\_onto\ f\ Ag= \quad _{iso} \tag{12}$$

The notation $fx\ j\ x\ y\ z{:}\,P\ x\ y\ zg$ is syntactic sugar for $ft{:}\,9x\ y\ z{:}\,t = x \wedge P\ x\ y\ zg$. The type $(\ ;\ )\,iso$ is well-defined because the characterizing set contains at least the class of bijections on the empty set. The relation $\ _{iso}$ is an equivalence relation on the set $f(f;A)\ j\ f\ A{:}\ inj\_onto\ f\ Ag$.

Requiring $inj\_onto\ f\ A$ in the definition of $\ _{iso}$ and also in the quotient construction (12) may seem redundant, but it simplifies congruence lemmas, as we will see in (13).

The basic operations on an isomorphism apply it to an element of its domain, determine its domain and range, invert it or compose it with another isomorphism. We define those operations in the following. For each of them, we must show that they are well-defined, i.e. that $\ _{iso}$ is a congruence with respect to their defining terms.

The application $f\ \ _= x$ of an isomorphism f to an element x of its domain is the application of a bijection of one of the representatives of f to x. If x is not in the domain of f, then the application yields the fixed element $\hat{\alpha} = ("x{:}\,false)$ of the range type of f: $f\ \ _= x = \hat{\alpha}$. The definition does not fix a representative of f to apply to x, but considers the bijections of all representatives of f. It maps them to functions that are identical to $\hat{\alpha}$ outside the domain of f, and chooses one of those functions to apply it to x. We define a function $fun_=$ that maps isomorphisms to HOL functions. The function *choice* chooses an arbitrary member of a set.

$$fun_= :: (\ \ ;\ \ )\,iso\ \ !\ \ \ \ \ \ !$$

$$fun_=\ f \overset{\text{def}}{=} choice\left(\bigcup_{p\,2\,\underline{iso}\ f}(\ \ (f;A){:}\,f(\ \ x{:}\ \text{if}\ \ x\ 2\ A\ \text{then}\,f\,x\ \text{else}\,\hat{a})g)\,p\right)$$

We use the infix notation $f\ \ _= x$ as a syntactic variant of the term $fun_=\ f\ x$, which is easy to define in Isabelle/HOL. The application is well-defined, because all members $(f;A)$ of an equivalence class representing an isomorphisms agree on A, i.e. the argument to *choice* is a singleton. In the following definitions, we use the representation function $\underline{iso}$ and the abstraction function $\overline{iso}$ that are induced by the type definition (12) (c.f. Sect. 2).

The domain $dom_=\ f$ of an isomorphism f is the second component of one of its representatives. The range $ran_=\ f$ is the image of the domain of f under f.

$$dom_= :: (\ \ ;\ \ )\,iso\ \ !\ \ \ \ set \qquad\qquad ran_= :: (\ \ ;\ \ )\,iso\ \ !\ \ \ \ set$$

$$dom_=\ f \overset{\text{def}}{=} choice\ \ (snd\ \ \grave{}\ \underline{iso}\ \ f\acute{})) \qquad\qquad ran_=\ f \overset{\text{def}}{=} fun_=\ f\ \grave{}dom_=\ f\acute{}$$

We obtain the inverse of an isomorphism by inverting its representing bijections on their domain.

$$inv_= :: (\ \ ;\ \ )\,iso\ \ !\ \ (\ \ ;\ \ )\,iso$$

$$inv_=\ f \overset{\text{def}}{=} \overline{iso}\left(\bigcup_{p\,2\,\underline{iso}\ f}(\ \ (f;A){:}\ \ \ \ _{iso}\ \ \grave{}f(inv\_on\ A\ f\,;f\,\grave{}A\acute{}))g\acute{})\,p\right)$$

To define the composition of two isomorphisms j and k, we consider all pairs $(f;A)$ representing j and $(g;B)$ representing k. A bijection representing the composition of j and k is the functional composition $g\ \ \ f$. The domain of the composed isomorphism is the inverse image of the part of the range $f\,\grave{}A\acute{}$ of j that is in the domain B of k.

$$\_\ {}^{\circ}_{9=}\ \_ :: [(\ \ ;\ \ )\,iso;(\ \ ;\gamma)\,iso]\ \ !\ \ (\ \ ;\gamma)\,iso$$

$$j\ {}^{\circ}_{9=}\ k \overset{\text{def}}{=} \overline{iso}\left(\bigcup_{p\,2\,\underline{iso}\ j}\ \bigcup_{q\,2\,\underline{iso}\ k}(\ \ (f;A){:}\,(\ \ (g;B){:}\right.$$

$$\left.\ _{iso}\ \ \grave{}f(g\ \ \ \ f;(inv\_on\ A\ f)\grave{}f\,\grave{}A\acute{}\ \backslash\ B\acute{}))g\acute{})\,q)\,p\right)$$

The relation $_{iso}$ is a congruence with respect to the defining function of $\_ \overset{\circ}{9}_{=} \_$ on representatives.

$$\begin{array}{l} congruent_2 \quad _{iso} \quad _{iso} \\ \qquad (\ p\,q : (\ (f;A) : (\ (g;B) : \ _{iso}\ (jf(g\ \ f; inv\_on\ A\ f\ (jf\ (jAj)\ \backslash\ Bj))gj))\ q)\ p) \end{array} \tag{13}$$

Lemma (13) illustrates the use of having two equivalence relations as parameters to $congruent_2$. The three occurrences of $_{iso}$ in (13) all have different types: the first is a relation on $(\ ;\ )\ isopair$, the second on $(\ ;\gamma)\ isopair$, and the third (in the body of the -term) on $(\ ;\gamma)\ isopair$.

The proof of (13) depends on the following fact about the composition of functions:

$$\frac{inj\_onto\ f\ A \qquad inj\_onto\ g\ B}{inj\_onto\ (g\ \ f)((inv\_on\ A\ f)\ (jf\ (jAj)\ \backslash\ Bj))} \tag{14}$$

Lemma (14) shows why it is convenient to restrict $_{iso}$ to injective functions (onto sets). Lemma (14) assumes injectivity of f on A. Because the two pairs $(f;A)$ and $(g;B)$ mentioned in (13) are bound variables, it is technically complex to assume that f and g are injective onto A and B, respectively – and thus to state the congruence proposition as a lemma. The local context of proofs that use that lemma guarantees that the arguments to the second parameter satisfy inj_onto, because the quotient construction in (12) considers only pairs that satisfy it. That knowledge, however, does not help us to state the congruence proposition as an independent lemma.

The application of isomorphisms is a “partial” concept. Therefore, an equality involving applications, such as $x\ \ 2\ dom_{=}\ f\ \ )\ \ inv_{=}\ f\ \ _{=}(f\ \ _{=}x) = x$, must be guarded by premises ensuring that the isomorphisms are applied to members of their domain only. The domain, composition, and inverses of isomorphisms have the nice algebraic properties one would expect. For example, $inv_{=}\ (inv_{=}\ f) = f$ holds unconditionally. This is a pay-off of the relatively complex type definition. The quotient construction with $_{iso}$ ensures that the operations on isomorphisms are “total” – as long as no applications of isomorphisms are involved.

7 Functors

To link a shallow embedding and a general theory by isomorphisms, the basic operations defined in the preceding section do not suffice. We additionally need mechanisms to “lift” isomorphisms on the sets of relevant parameter data to isomorphisms on (sets of elements of) more complex types. In the example of Sect. 3, consider the input parameters of an operation that is included at two different positions in two class specifications representing the same class. The set of its input parameters are thus isomorphic to the ranges of injections into the types of input parameters of the two class specifications. If we wish to express formally that the two specifications are isomorphic, then we must, first, construct isomorphisms from the injections in the two sum types, and second, map those isomorphisms to an isomorphism that maps the first class specification to the second. In the present section, we define functors mapping isomorphisms to isomorphisms on complex types. We consider product types, function spaces, ranges of injections, and type definitions.

Products. Given two isomorphisms, f and g, it is easy to construct the product isomorphism $f\ \ _{=}\ g$ whose domain is the Cartesian product of their domains: the product isomorphism relates two pairs if the two parameter isomorphisms relate the components of the pairs. The basic functions on isomorphisms distribute over product isomorphisms in the way one expects. The following fact illustrates that for the application of a product isomorphism.

$$8x{:}x\ \ 2\ dom_{=}(f\ \ \ _{=}\ g)\ \ )\ \ f\ \ \ _{=}\ g\ \ \ _{=}\ x = (f\ \ \ _{=}\ fst\ x; g\ \ \ _{=}\ snd\ x) \tag{15}$$

Function Spaces. We construct isomorphisms of – total – functions from isomorphisms that relate subsets of their domain and range types. Given an isomorphism f with domain A and range A^0, and an isomorphism g with domain B and range B^0, we wish to construct an isomorphism of the functions mapping A to B and the functions mapping A^0 to B^0. Speaking set-theoretically, the construction is a functor from the category of sets to the category of set-theoretic functions (where we consider only isomorphisms in both categories). Thus, if h is a function from A to B, we construct a function k from A^0 to B^0 by making the functions on the sets A, A^0, B, and B^0 commute.

However, the need to work with total HOL-functions complicates the definition, because we must define the value of function applications outside their "domains". The HOL-function spaces $\ \ !\ \ $ and $\ ^0\ !\ \ ^0$, in general, are not isomorphic. Therefore, we consider only functions that are constant on the complements of A and B (if the complements are non-empty). To be able to control the values of the functions in the domain of a lifted isomorphisms in that way, we supply the image values $c\ ::\ \ $ of h and $d\ ::\ \ ^0$ of k outside their domains as parameters to the functor. The set $\overline{A}$ is the complement of A.

$$lift_{=} :: [\ \ ;\ \ ^0; (\ \ ;\ \ ^0)\ iso; (\ \ ;\ \ ^0)\ iso]\ \ !\ \ (\ \ !\ \ ;\ \ ^0\ !\ \ ^0)\ iso$$

$$\begin{aligned} lift_{=}\ c\ d\ f\ g \overset{\text{def}}{=} \overline{iso} \Bigg(\bigcup_{p\,2\,\underline{iso}\ f}\ \bigcup_{q\,2\,\underline{iso}\ g} (\ \ (f;A).(\ \ (g;B).\ & \\ {}_{iso}\ (jf((\ \ hx.\ \text{if}\ \ x\ 2\ f(jAj)\ \text{then}\ (g\ \ \ h\ \ \ (inv\_on\ A\ f))x & \\ \text{else}\ d); & \\ fh\ j\ h.h(jAj)\ \ \ \ B\ \wedge\ h(j\overline{A}j)\ 2\ f\varnothing; fcggg)gj)\ )q)p & \Bigg) \end{aligned}$$

The following lemmas establish equalities for the domain, inverse, and the application of the lifting of isomorphisms to function spaces.

$$dom_{=}(lift_{=}\ c\ d\ f\ g) = fh\ \ j\ h.h\ (jdom_{=}\ fj)\ \ \ \ dom_{=}\ g\ \wedge\ h\ (j\overline{dom_{=}\ f}j)\ \ 2\ \ f\varnothing; fcggg \tag{16}$$

$$inv_{=}(lift_{=}\ c\ d\ f\ g) = lift_{=}\ d\ c\ (inv_{=}\ f)(inv_{=}\ g) \tag{17}$$

$$\frac{h\ (jdom_{=}\ fj)\ \ \ \ dom_{=}\ g\ \ \ \ h\ (j\overline{dom_{=}\ f}j)\ \ 2\ \ f\varnothing; fcgg}{lift_{=}\ c\ d\ f\ g\ \ \ _{=}\ h = (\ \ a^0.\ \text{if}\ \ a^0\ 2\ ran_{=}\ f\ \ \text{then}\ g\ \ \ _{=}\ h(inv_{=}\ f\ \ \ _{=}\ a^0)\ \text{else}\ d)} \tag{18}$$

Proving those lemmas with Isabelle/HOL is remarkably complex. Although the properties of lifting isomorphisms are not mathematically deep theorems, we must formulate them carefully to account not only for the – mostly obvious – "normal" cases that deal with the relation of the involved functions on the domains and ranges of the lifted isomorphisms, but we must also consider their “totalization” on the complements of those sets. The parts of the proofs dealing with the complements of domains and ranges are the technically difficult ones. Here, the prover needs guidance to find appropriate witnesses, and to establish contradictions. The need to apply the extensionality rule of functions further reduces the degree of automation, because automatically applying that rule would lead to an explosion of the search space.

Ranges of Injections. Two embeddings of a type into types and , such as injections of a type in two different sum types in Sect. 3, induce an isomorphism on the ranges of the embeddings. If two functions f and g are injective, then the ranges of f and g are isomorphic and we can construct a canonic bijection $h = g \quad f^{-1}$ between the two ranges that makes the diagram commute. Generalizing to possibly non-injective functions, we use the maximal set onto which a function f is injective to determine the bijection. That set is the union of all sets onto which f is injective: $(\bigcup (Collect\ (inj\_onto\ f)))$. If A is the maximal subset of onto which both, f and g, are injective, then the images of A under f and g are isomorphic. A bijection between the images is the composition of g with the inverse of f onto the set on which f is injective, $g \quad (inv\_on\ (\bigcup (Collect\ (inj\_onto\ f)))\ f)$. Capturing this idea formally, we define a function $(\_ \cdot =^{\&} \_)$ that maps two functions with the same domain type to an isomorphism of their range types.

$$(\_ \cdot =^{\&} \_) :: [\quad ! \quad ; \quad ! \ y]\ !\ (\ ;y)\ iso$$

$$f \cdot =^{\&} g \overset{def}{=} \overline{iso}\ (\quad _{iso}\ (\!|f(g \quad (inv\_on\ (\bigcup (Collect\ (inj\_onto\ f)))\ f);$$
$$f\ (\!|\bigcup (Collect\ (inj\_onto\ f))\ \backslash\ \bigcup (Collect\ (inj\_onto\ g)))|\!)\ )g|\!)\ )$$

We are interested primarily in range isomorphisms $f \cdot =^{\&} g$ where both, f and g, are *injective* on the whole type. Nevertheless, we must define $f \cdot =^{\&} g$ for arbitrary f and g. The following lemmas show the interesting properties of injection isomorphisms that are constructed from injective functions.

$$\frac{inj\ f \qquad inj\ g}{dom_{=}\ (f \cdot =^{\&} g) = range\ f} \qquad \frac{inj\ f \qquad inj\ g \qquad x\ \ 2\ range\ f}{(f \cdot =^{\&} g)\ \ _{=}\ x = g(Inv\ f\ x)}$$

$$\frac{inj\ f \qquad inj\ g}{ran_{=}\ (f \cdot =^{\&} g) = range\ g} \qquad \frac{inj\ f \quad inj\ g}{inv_{=}\ (f \cdot =^{\&} g) = g \cdot =^{\&} f}$$

Type Definitions. We saw in Sect. 2 that a type definition introduces three constants along with the new type: the representing set R, the abstraction function Abs, and the representation function Rep. The functions Abs and Rep establish a bijection between R and the new type. The axioms (1), (2), and (3) capture that property. We introduce a predicate is_typabs that characterizes the triples $(R; Rep; Abs)$ that establish a type definition according to (1), (2), and (3).

Consider two type definitions, $is\_typabs\ D\ \_\ ^{-}$ and $is\_typabs\ R\ \_\ ^{-}$. Let h be an isomorphism mapping elements of D to elements of R. It induces an isomorphism

$typ_{=}$ _ h _ on the types and that maps an element a :: to an element $(c$ :: $) =$ $(^{-}(h\ \ _{=}$ (_ $a))$, where we use the fact that ¯ is the inverse of _ on R. We do *not* require the domain of h to comprise the entire set D. Therefore, the domain of $typ_{=}$ _ h _, in general, is not the entire type but the subset of that is the co-image of the intersection of the domain of h and D under the abstraction function ¯.

To define the function $typ_{=}$, we first introduce injection isomorphisms, which are a special case of range isomorphisms, where the first argument is the identity function:

$$inj_{=}\ f \overset{\text{def}}{=} (\quad u{:}\,u)\ ^{\cdot}\ =^{\&}\ f$$

The injection isomorphism of a representation function, such as _, maps the entire type to its representing set, because the representation function is injective. The lifting of an isomorphism h via two type definitions, therefore, is the composition of the injection isomorphism of the representation function whose range type is the domain type of h with h and the inverse of the injection isomorphism of the representation function whose range type is the range of h.

$$typ_{=}\ ::\ [\quad !\quad ^{0};(\ ^{0};\ ^{0})\,iso;\quad !\quad ^{0}]\ !\ (\ ;\ )\,iso$$

$$typ_{=}\ Rl\ h\ Rr \overset{\text{def}}{=} (inj_{=}\ Rl)\ ^{\circ}_{9=}\ (h\ ^{\circ}_{9=}\ (inv_{=}\ (inj_{=}\ Rr)))$$

It is advantageous to use the injection isomorphisms of the representation functions in the definition of $typ_{=}$, because composing them with h appropriately constrains the domain of the lifted isomorphism. In a typical application, the representation functions Rl and Rr are instances of the same polymorphic representation function R, because the lifted isomorphism relates instances of a single polymorphic type.

8 Predicate Isomorphisms

The components of a class specification in the example of Sect. 3 are basically predicates (the operation specifications in the method suite are also predicates). To construct an isomorphism between class specifications from isomorphisms on their parameter data, i.e. the constants, states, inputs, and outputs, we need to lift those isomorphisms to isomorphisms on predicates. Conceptually, this is an easy application of the lifting of isomorphisms to function spaces. Establishing the necessary theorems in Isabelle/HOL, however, turned out to harder than expected.

Consider an isomorphism f that relates two sets A and A^{0} of some arbitrary types and ^{0}. The lifting isomorphism of f and the identity isomorphism $id_{=}$ on *bool* maps a predicate h on to a predicate k on ^{0}. We assume that the domain of f comprises the extension *Collect* h of h. Under that assumption, we require the lifting isomorphism to preserve the extension of predicates, because otherwise the lifting isomorphism would not preserve the information provided by the predicates in its domain. (For the components of class schemas, this requirement ensures that specifications such as the constraints on the constants of the objects of a class are preserved.) The lifting isomorphism must map each predicate h in its domain to a predicate k that is constantly false outside the range of f. Providing *false* as the first two arguments to the lifting function $lift_{=}$, we enforce both, the assumption on the functions in the domain of the lifting isomorphism, and the requirement on the functions in its range. These considerations lead us to define lifting isomorphisms of (unary) predicates as follows:

$$"_{=} :: (\; ; \; ^{0})\, iso \; ! \; (\; ! \; bool; \; ^{0} \; ! \; bool)\, iso$$

$$"_{=} f \overset{\text{def}}{=} lift_{=}\; false\; false\; f\; id_{=}$$

The domain of the lifting of an isomorphism f to unary predicates is the set of all predicates P whose extensions are in the power set of the domain of f, i.e. a predicate P is in the domain of $"_{=} f$ if and only if the extension of P is a subset of the domain of f. For all P in the domain of $"_{=} f$, the extensions of P and of $"_{=} f \quad _{=} P$ are isomorphic (and related by f). Consequently, the predicates P and $"_{=} f \quad _{=} P$ are equivalent modulo renaming their parameters by f.

$$\frac{Collect\; P \qquad dom_{=}\, f}{P \;\; 2\; dom_{=}\, ("_{=} f)} \qquad\qquad \frac{Collect\; P \qquad dom_{=}\, f \quad x \;\; 2\; dom_{=}\, f}{P\, x \; = \; ("_{=} f \quad _{=} P)(f \quad _{=} x)}$$

$$\frac{P \;\; 2\; dom_{=}\, ("_{=} f)}{Collect\; P \qquad dom_{=}\, f} \qquad\qquad \frac{Collect\; P \qquad dom_{=}\, f \quad ("_{=} f \quad _{=} P)\, x}{x \;\; 2\; ran_{=}\, f}$$

Conceptually, it is easy to extend the lifting of unary predicates to predicates with several parameters. For binary predicates, we define $\uparrow\uparrow_{=} f\; g$.

$$\uparrow\uparrow_{=} :: [(\; ; \; ^{0})\, iso; (\; ; \; ^{0})\, iso] \; ! \; ([\; ; \;] \; ! \; bool; [\; ^{0}; \; ^{0}] \; ! \; bool)\, iso$$

$$\uparrow\uparrow_{=} f\; g \overset{\text{def}}{=} lift_{=}\, (\quad x{:}\, false)\, (\quad x{:}\, false)\, f\, ("_{=} g)$$

The lifting operation $\uparrow\uparrow_{=}$ and the similarly defined $\uparrow\uparrow\uparrow_{=}$ for ternary predicates have properties similar to the ones of $"_{=}$ shown before. The premises of those rules, however, require the domains of the parameter isomorphisms to be supersets of *projections* of the extensions of the involved binary or ternary predicates. For example, the rule establishing the equivalence of a predicate P and its image under $\uparrow\uparrow_{=} f\; g$ reads as follows:

$$\frac{\begin{array}{c} fx \; j\; x\; y{:}\, P\, x\, yg \qquad dom_{=}\, f \\ fy \; j\; x\; y{:}\, P\, x\, yg \qquad dom_{=}\, g \\ x \;\; 2\; dom_{=}\, f \quad y \;\; 2\; dom_{=}\, g \end{array}}{P\, x\, y \; = \; (\uparrow\uparrow_{=} f\; g \quad _{=} P)\,(f \quad _{=} x)\,(g \quad _{=} y)}$$

Facts about $\uparrow\uparrow_{=}$ and $\uparrow\uparrow\uparrow_{=}$ are remarkably hard to establish in Isabelle. Whereas the proofs about $"_{=}$ involve few trivial interactions, the proofs of the corresponding lemmas about $\uparrow\uparrow_{=}$ and $\uparrow\uparrow\uparrow_{=}$, in particular the ones concerning containment in the domains of the lifted isomorphisms and membership in the ranges of the lifted isomorphisms require considerable guidance to constrain the search space of automatic tactics. We needed to supply witnesses for most of the existential propositions, because the existential quantifiers are nested, and the automatic tactics try to find witnesses for the outermost quantifiers first. When proving theorems about projections to the second or third argument of predicates, that proof strategy leads to a combinatorial explosion of cases. In other lemmas, several premises stating subset relations lead to a combinatory explosion. To make use of the premises, we had to supply appropriate members of those sets explicitly, because the automatic proof tactics would not find them automatically. Some proofs led to subgoals with premises that are subset relations between sets of functions, or involved equalities between predicates. To prove those subgoals, we had to supply functional witnesses or use the extensionality rule of functions. The implemented tactics would not succeed in doing so automatically.

9 Abstract and Concrete Reasoning About Classes

With the theory of isomorphisms developed in the preceding sections, we can now define a lifting of isomorphisms on the parameter data of a class specification. We lift isomorphisms Ci, Si, Ii, and Oi on the parameter data of a class specification to isomorphisms on the component predicates. The product of those isomorphisms (lifted via the type definition of class specifications) yields an isomorphism on class specifications.

$$\begin{aligned} *^{C}_{=} &:: [(\ ;\ {}^{0})\,iso; (\ ;\ {}^{0})\,iso; (\ ;\ {}^{0})\,iso; (!\,; !^{0})\,iso] \\ &\quad !\ ((\ ;\ ;\ ;\ ; !)\,Class; (\ ;\ {}^{0};\ {}^{0};\ {}^{0}; !^{0})\,Class)\,iso \end{aligned}$$

$$\begin{aligned} *^{C}_{=}\ Ci\ Si\ Ii\ Oi \overset{\text{def}}{=} \Big(\textbf{let}\ C^{0} &= (''_{=}\ Ci); \\ S^{0} &= (\uparrow\uparrow_{=}\ Ci\ Si); \\ I^{0} &= (\uparrow\uparrow_{=}\ Ci\ Si); \\ M^{0} &= (map_{=}\ (*^{M}_{=}\ Ci\ Si\ Ii\ Oi)); \\ H^{0} &= (\uparrow\uparrow\uparrow_{=}\ Ci\ Si\ Si) \\ \textbf{in}\ typ_{=}\ \underline{Class}\ (C^{0}\ \ {}_{=}\,S^{0}\ \ {}_{=}\,I^{0}\ \ {}_{=}\,M^{0}\ \ {}_{=}\,H^{0})\,\underline{Class}\Big) \end{aligned}$$

The functor $*^{M}_{=}$ lifts isomorphisms to methods. The functor $map_{=}$ maps an isomorphism to one on finite mappings: the resulting isomorphism relates mappings with identical domains and isomorphic ranges.

What is the domain of an isomorphism on class specifications produced by such a lifting? A class specification Cls is in the domain of $*^{C}_{=}\ Ci\ Si\ Ii\ Oi$ if the relevant parameter data, i.e. the data for which one of the component predicates of Cls is true, is in the domain of the appropriate parameter isomorphism. For example, the set of admissible states of Cls must be a subset of the domain of Si.

The theory of isomorphisms precisely captures the property of class specifications representing the same class. It also allows us to unify the types of an arbitrary finite number of class specifications. Consider, for example, the two class specifications $A :: (\ ;\ {}_{1};\ {}_{1};\ {}_{1}; !_{1})\ Class$ and $B :: (\ ;\ {}_{2};\ {}_{2};\ {}_{2}; !_{2})\ Class$. Similar to the construction of a method suite from operation specifications in Sect. 3, we map both class specifications to the type of class specifications $(\ ;\ {}_{1} + \ {}_{2};\ {}_{1} + \ {}_{2};\ {}_{1} + \ {}_{2}; !_{1} + !_{2})\ Class$ whose parameter types are the sums of the parameter types of A and B, where Inl and Inr are the usual injection functions into a sum type:

$$\begin{aligned} A^{0} &= *^{C}_{=}\ (inj_{=}\ Inl)\,(inj_{=}\ Inl)\,(inj_{=}\ Inl)\,(inj_{=}\ Inl)\ \ {}_{=}\,A \\ B^{0} &= *^{C}_{=}\ (inj_{=}\ Inr)\,(inj_{=}\ Inr)\,(inj_{=}\ Inr)\,(inj_{=}\ Inr)\ \ {}_{=}\,B \end{aligned}$$

The use of such an embedding of class specifications into the same type lies in the possibility to derive abstract, general theorems about classes and apply them to the specifications of a concrete system. Consider, for example, the refinement relation $A \; v^{R}_{(\; ; \;)} \; B$ on class schemas A and B (of possibly different types), which we define in [18]. The predicate R relates the object states of A and B, maps the method identifiers of A to the ones of B, and is a family of isomorphisms on the input/output types of the method suites of A and B that relates the valid I/O of the methods of A to the valid I/O of the methods of B that refine the ones of A (as indicated by).

Proving $A \; v^{R}_{(\; ; \;)} \; B$ for concrete classes A and B amounts to reducing the proposition to verification conditions on the component schemas of A and B, which are stated in the $HOL\text{-}Z$ encoding of Z. Tactics accomplish that reduction uniformly. They also synthesize the isomorphisms in automatically. Because $HOL\text{-}Z$ is a shallow embedding, the proofs of those verification conditions are “ordinary” proofs in HOL that can make use of the standard proof procedures implemented in Isabelle.

Refinement proofs for concrete classes are costly. It is therefore advantageous to derive theorems about the refinement relation in general such that they can be applied with relatively little effort to concrete specifications, instead of implementing tactics that derive an instance of a general theorem anew for each concrete instance. In the context of a non-trivial software development, two properties of refinement, transitivity and compositionality, are of particular importance, because they allow developers to refine classes independently of their application context in a step-wise manner.

For our definition of refinement, it is possible to state the transitivity proposition such that the three involved classes have arbitrary, possibly different types. If we wish to define the reflexive and transitive closure v , however, type constraints force all involved classes to have the same type.

Compositionality ensures that classes can be refined within a context of their use. For example, it is desirable that replacing the class of an aggregated object by a refining class yields a refinement of the aggregating context also. Formally, this means that the context F is monotonic with respect to refinement:

$$A \; v^{R}_{(\; ; \;)} \; B \;) \; F(A) \; v^{R'}_{(id;id)} \; F(B) \tag{19}$$

Stating (19) in HOL forces A and B to have the same type. The context F is a HOL-function that, in general, refers to the methods of its parameter class. Because method invocation is one of the points where the deep encoding of class specifications is linked with the shallow embedding of $HOL\text{-}Z$, method invocation must appropriately inject input and output parameters into the sum types we discussed in Sect. 3. Those injections induce a specific structure on the type of the method suite of the parameter of F and thus on the types of A and B. Furthermore, to define R^{0} in terms of R, the state of the resulting classes $F(A)$ or $F(B)$ must have a particular structure: we need to know where the aggregated object of class A (or B) resides in the tuple of state components. “Without loss of generality” we may fix that to be the first component, because an appropriate isomorphism on class specifications can change the order of state components as needed for a particular concrete specification.

We can state sufficient conditions on the context F and prove a proposition such as (19) for arbitrary classes A and B – of the *same* type. Isomorphisms allow us to apply that theorem to classes of *different* types, which arise from the shallow encoding of concrete specifications in $HOL\text{-}Z$. Thus, isomorphisms play a vital role in linking that shallow encoding with the deep definitions of object-oriented concepts and the theory about them.

10 Conclusions

We are not aware of other work addressing a formal theory of isomorphic sets in a typed logic. As we saw, the foundational definitions are quite complex, because the concept of an isomorphism between *sets* is inherently partial, whereas the world of HOL is total. The theory we presented "hides" that partiality for all operations except the application of isomorphisms. For those, domain considerations are necessary. In concrete applications of the theory, however, these can often be reduced to trivial verification conditions. Müller and Slind [11] survey methods of representing partial functions in typed logics, in particular in Isabelle/HOL. Those methods leave the domain of functions implicit, e.g., by returning a special value if a function is applied outside its domain. Our definition of isomorphisms explicitly contains the domain as a set, because we frequently need to consider the domain of an isomorphism to assess whether it is suitable to be applied to a given set of data. Leaving the domain implicit in the definition would therefore not make reasoning simpler.

The theory of isomorphisms presented in this paper emerged from the combination of *HOL-Z* with a general theory of object-orientation that we sketched in Sections 3 and 9. The application of the combined theory to the Eiffel libraries, which we mentioned in Sect. 3, shows its suitability for practical applications. In that application, isomorphisms relate the I/O data of different class specifications. Tactics uniformly construct those isomorphisms. Once the appropriate lifting operations are defined, the construction is simple.

We believe that the approach of combining a shallow embedding with a general theory is useful for other theories, too. For example, one could augment the theory of CSP in Isabelle/HOL [20] with a shallow embedding of data specifications. This avoids representing complex theorems by tactic code only, as it is necessary in a "completely shallow" approach, such as [5] and most other approaches aiming at practical applications that we know of. Moreover, general purpose theories in HOL, such as a theory of (typed) relations, become immediately applicable to the combined theory.

Apart from linking "shallow" and "deep", isomorphisms also admit to switch between a "type-view" and a "set-view" of data. Based on the isomorphism between a type and its representing set, predicate isomorphisms directly relate properties of the elements of the type to properties of the members of the representing set. Exploiting that relation may help to reduce the sometimes annoying technical overhead of "lifting" functions (and their properties) that are defined on the representing set to corresponding functions on the type.

Acknowledgments. Thanks to the referees for their constructive comments and for their patience with the first version of the paper.

References

[1] J. Bowen and M. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5–6):269–276, 1995.

[2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[3] M. Gordon. Set theory, higher order logic or both? In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 191–201. Springer-Verlag, 1996.

[4] M. J. C. Gordon and T. M. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[5] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin, editor, *Programming Languages and Systems — 7th European Symposium on Programming, ESOP'98*, LNCS 1381, pages 105–121. Springer-Verlag, 1998.

[6] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[7] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher-Order Logics*, LNCS 1125, pages 283–298. Springer-Verlag, 1996.

[8] L. Lamport and L. C. Paulson. Should your specification language be typed? http://www.cl.cam.ac.uk/users/lcp/papers/lamport-paulson-types.ps.gz, 1997.

[9] B. Meyer. *Reusable Software*. Prentice Hall, 1994.

[10] O. Müller. I/O automata and beyond: temporal logic and abstraction in Isabelle. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, LNCS 1479, pages 331–348. Springer-Verlag, 1998.

[11] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–651, 1997.

[12] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics — 11th International Conference, TPHOLs'98*, LNCS 1479, pages 349–366. Springer-Verlag, 1998.

[13] T. Nipkow. More Church/Rosser proofs (in Isabelle/HOL). In *Automated Deduction — CADE 14*, LNCS 1104, pages 733–747. Springer-Verlag, 1996.

[14] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, 1993.

[15] L. C. Paulson. *Isabelle – A Generic Theorem Prover.* LNCS 828. Springer-Verlag, 1994.

[16] T. Santen. A theory of structured model-based specifications in Isabelle/HOL. In E. L. Gunter and A. Felty, editors, *Proc. International Conference on Theorem Proving in Higher Order Logics*, LNCS 1275, pages 243–258. Springer-Verlag, 1997.

[17] T. Santen. On the semantic relation of Z and HOL. In J. Bowen and A. Fett, editors, *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 96–115. Springer-Verlag, 1998.

[18] T. Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification*. PhD thesis, Fachbereich Informatik, Technische Universität Berlin, Germany, 1999. forthcoming.

[19] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[20] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer-Verlag, 1997.

Polytypic Proof Construction

Holger Pfeifer[1] and Harald Rueß[2]

[1] Universität Ulm, Fakultät für Informatik, D-89069 Ulm, Germany
pfeifer@informatik.uni-ulm.de
[2] SRI International, Computer Science Laboratory, 333 Ravenswood Ave.
Menlo Park, CA, 94025, USA
ruess@csl.sri.com

Abstract. This paper deals with formalizations and verifications in type theory that are abstracted with respect to a class of datatypes; i.e *polytypic* constructions. The main advantage of these developments are that they can not only be used to define functions in a generic way but also to formally state polytypic theorems and to synthesize polytypic proof objects in a formal way. This opens the door to mechanically proving many useful facts about large classes of datatypes *once and for all*.

1 Introduction

It is a major challenge to design libraries for theorem proving systems that are both sufficiently complete and relatively easy to use in a wide range of applications (see e.g. [6, 26]). A library for abstract datatypes, in particular, is an essential component of every proof development system. The libraries of the Coq [1] and the Lego [13] system, for example, include a number of functions, theorems, and proofs for common datatypes like natural numbers or polymorphic lists. In these systems, myriads of mostly trivial developments are carried out separately for each datatype under consideration. This increases the bulk of proving effort, reduces clarity, and complicates lemma selection. In contrast, systems like Pvs [20] or Isabelle [22] support an infinite number of datatypes by using meta-level functions to generate many standard developments from datatype definitions. Pvs simply uses an extension of its implementation to generate axiomatized theories for datatypes including recursors and induction rules, while Isabelle's datatype extension includes tactics to prove these theorems. Both the Pvs and the Isabelle approach usually work well in practice. On the other hand, meta-level functions must be executed separately for each datatype under consideration, and construction principles for proofs and programs are operationalized and hidden in meta-level functions which are encoded in the implementation language of the proof system.

In this paper we propose techniques for building up library developments from a core system without resorting to an external meta level. Moreover, proof objects are constructed explicitly and developments from the library can be used by simply instantiating higher-order quantifiers. Hereby, we rely on the concept of *polytypic abstraction*.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 55–72, 1999.

The *map* functional on the list datatype L illustrates the concept of polytypic abstraction exceptionally well. Applying this functional to a function f and a source list l yields a target list obtained by replacing each element a of l with the value of $f(a)$, thereby preserving the structure of l. The type of *map* in a Hindley-Milner type system, as employed by current functional programming languages, is abstracted with respect to two type variables A and B:

$$map : \forall A, B.(A \to B) \to L(A) \to L(B)$$

Thus, *map* is a *polymorphic* function. The general idea of the *map* function of transforming elements while leaving the overall structure untouched, however, is not restricted to lists and applies equally well to other datatypes. This observation gives rise to a new notion of polymorphism, viz. *polytypy*,[1] for defining functions uniformly on a class of (parameterized) datatypes $\mathcal{T}$:

$$map : \forall \mathcal{T}.\forall A, B.(A \to B) \to \mathcal{T}(A) \to \mathcal{T}(B)$$

Notice that the notion of polytypy is completely orthogonal to the concept of polymorphism, since every "instance" of the family of polytypic *map*-functions is polymorphic. Many interesting polytypic functions have been identified and described in the literature [15, 25, 17, 10, 11, 16, 27], and concepts from category theory have proven especially suitable for expressing polytypic functions and reasoning about them. In this approach, datatypes are modeled as initial objects in categories of functor-algebras [14], and polytypic constructions are formulated using initiality without reference to the underlying structure of datatypes.

The concept of polytypy, however, is not restricted to the definition of polytypic functions solely, but applies equally well to other entities of the program and proof development process like specifications, theorems, or proofs. Consider, for example, the *no confusion* theorem. This theorem states that terms built up from different constructors are different. It is clearly polytypic, since it applies to all initial datatypes. In the following, we examine techniques for expressing polytypic abstraction in type theory. These developments can not only be used to polytypically define functions but also to formally state polytypic theorems and to interactively develop polytypic proofs using existing proof editors. Thus, formalization of polytypic abstraction in type theory opens the door to proving many useful facts about large classes of datatypes *once and for all* without resorting to an external meta-language.

The paper is structured as follows. The formal setting of type theory is sketched in Section 2, while Section 3 includes type-theoretic formalizations of some basic notions from category theory that are needed to specify, in a uniform way, datatypes as initial objects in categories of functor algebras. Furthermore, Section 3 contains generalizations of the usual reflection and fusion theorems for recursors on initial datatypes — as stated, for example, in [2] — to recursors of dependent type, which correspond to structural induction schemes. These developments are polytypically abstracted for the *semantically* characterized class of

[1] Sheard [25] calls these algorithms *type parametric*, Meertens [15] calls them *generic*, and Jay and Cockett [9] refer to this concept as *shape polymorphism*.

initial datatypes. This notion of semantic polytypy, however, is not appropriate in many cases where inspection of the form of the definition of a datatype is required. Consequently, in Section 4, we develop the machinery for abstracting constructions over a certain *syntactically* specified class of datatypes. Since the focus is on the generation of polytypic proofs rather than on a general formalization of inductive datatypes in type-theory we restrict ourselves to the class of parameterized, polynomial (sum-of-products) datatypes in order to keep the technical overhead low. The main idea is to use representations that make the internal structure of datatypes explicit, and to compute type-theoretic specifications for all these datatypes in a uniform way. This approach can be thought of as a simple form of *computational reflection* (e.g. [23,29]). Developments that are abstracted with respect to a syntactically characterized class of datatypes are called *syntactically polytypic* in the following. We demonstrate the expressiveness of syntactic polytypy with a mechanized proof of the bifunctoriality property for the class of polynomial datatypes and a polytypic proof of the *no confusion* theorem. Finally, Section 5 concludes with some remarks.

The constructions presented in this paper have been developed with the help of the Lego [13] system. For the sake of readability we present typeset and edited versions of the original Lego terms and we take the freedom to use numerous syntactic conventions such as infix notation and pattern matching.

2 Preliminaries

Our starting point is the *Extended Calculus of Constructions (ECC)* [12] enriched with the usual inductive datatypes. We sketch basic concepts of this type theory, fix the notation, and discuss the treatment of datatypes in type theory. More interestingly, we introduce n-ary (co-)products and define functions on generalized (co-)products by recursing along the structure of the descriptions of these types. This technique has proven to be essential for the encoding of syntactic polytypy in Section 4.

The type constructor $\Pi x : A.B(x)$ is interpreted as the collection of dependent functions with domain A and codomain $B(a)$ with a the argument of the function at hand. Whenever variable x does not occur free in $B(x)$, $A \to B$ is used as shorthand for $\Pi x : A.B(x)$; as usual, the type constructor $\to$ associates to the right. λ-abstraction is of the form $(\lambda x : A.M)$ and abstractions like $(\lambda x : A, y : B.M)$ are shorthand for iterated abstractions $(\lambda x : A.\ \lambda y : B.M)$. Function application associates to the left and is written as $M(N)$, as juxtaposition $M\ N$, or even in subscript notation M_N. Types of the form $\Sigma x : A.B(x)$ comprise dependent pairs $(.,.)$, and π^1, π^2 denote the projections on the first and second position, respectively. Sometimes, we decorate projections with subscripts as in $\pi^1_{A,B}$ to indicate the source type $\Sigma x : A.B(x)$. Finally, types are collected in yet other types *Prop* and $Type_i$ $(i \in \mathbb{N})$. These universes are closed under the type-forming operations and form a fully cumulative hierarchy [12]. Although essential to the formalization of many programming concepts, universes are tedious to use in practice, for one is required to make specific choices of universe levels.

For this reason, we apply | carefully, without introducing inconsistencies | the *typical ambiguity* convention [7] and omit subscripts i of type universes $Type_i$.

Definitions like $c(x_1 : A_1;:::x_n : A_n) : B \quad ::= \quad M$ are used to introduce a name c for the term M that is (iteratively) abstracted with respect to x_1 through x_n. Bindings of the form $x \; j \; A$ are used to indicate parameters that can be omitted in function application. Systems like Lego are able to infer the hidden arguments in applications automatically (see [13]).

Using the principle of *propositions-as-types*, the dependent product type $x : A{:}B(x)$ is interpreted as logical universal-quantification and if $M(a)$ is of type $B(a)$ for all $a : A$ then $x : A{:}M(x)$ is interpreted as a proof term for the formula $x : A{:}B(x)$. It is possible to encode in ECC all the usual logical connectives ($>$, $?$, $\wedge$, $\_$, $:$, $)$) and quantifiers (8, 9, 9^1) together with a natural-deduction style calculus for a higher-order constructive logic. Leibniz equality ($=$) identifies terms having the same properties. This equality is intensional in the sense that $a \; = \; b$ is inhabited in the empty context if and only if a and b are convertible; i.e. they are contained in the least congruence $'$ generated by -reduction. Constructions in this paper employ, besides Leibniz equality, a (restricted) form of extensional equality (denoted $\doteq$) on functions.

$$: \doteq :(A;B \; j \; Type)(f;g : A \; ! \; B) : Prop \quad ::= \quad 8x : A{:}f(x) \; = \; g(x)$$

Inductive datatypes can be encoded in type theories like ECC by means of impredicative quantification [3]. For the well-known imperfections of these encodings | such as noninhabitedness of structural induction rules | however, we prefer the introduction of datatypes by means of formation, introduction, elimination, and equality rules [18,4,21]. Consider, for example, the extension of type theory with (inductive) products. The declared constant $: \quad :$ forms the product type from any pair of types, and pairing $(:;:)$ is the only constructor for this newly formed product type.

$$: \quad : : Type \; ! \; Type \; ! \; Type \qquad (:;:) : \quad A;B \; j \; Type{:}A \; ! \; B \; ! \; (A \quad B)$$

The type declarations for the product type constructor and pairing represent the formation and introduction rules of the inductive product type, respectively. These rules determine the form of the elimination and equality rules on products.

$$\begin{aligned} elim \quad : \quad & A;B \; j \; Type;\; C : (A \quad B) \; ! \; Type{:} \\ & (\quad a : A;b : B{:}C(a;b)) \; ! \quad x : (A \quad B){:}C(x) \end{aligned}$$

Elimination provides a means to construct proof terms (functions) of propositions (types) of the form $x : (A \quad B){:}C(x)$. The corresponding equality rule is specified in terms of a left-to-right rewrite rule: $elim_C \; f \; (a;b) \rightsquigarrow f(a)(b)$. It is convenient to specify a *recursor* as the non-dependent variant of elimination in order to define functions such as the first and second projections.

$$\begin{aligned} rec \;\; (A;B;C \; j \; Type) \quad & ::= \quad elim_{\;\_:A \quad B:C} \\ fst : (A \quad B) \; ! \; A \quad & ::= \quad rec \;\; (\quad x : A;y : B{:}x) \\ snd : (A \quad B) \; ! \; B \quad & ::= \quad rec \;\; (\quad x : A;y : B{:}y) \end{aligned}$$

Moreover, the (overloaded) $:\quad :$ functional is a bifunctor (see also Def. 2) and plays a central role in categorical specifications of datatypes; it is defined by means of the *split* functional $hf;gi$.

$$\begin{aligned}
&h:;:i(C\ j\ Type)(f : C\ !\ A;\ g : C\ !\ B) :\ C\ !\ (A \quad B)\\
&\qquad ::= \quad x : C{:}(f(x);\ g(x))\\
&:\quad :(A;B;C;D\ j\ Type)(f : A\ !\ C;\ g : B\ !\ D) : (A \quad B)\ !\ (C \quad D)\\
&\qquad ::=\ hf \quad fst_{A;B};\ g \quad snd_{A;B}\,i
\end{aligned}$$

The specifying rules for coproducts $A+B$ with injections $inl_{A;B}(a)$ and $inr_{A;B}(b)$ are dual to the ones for products. Elimination on coproducts is named $elim^{+}$ and its non-dependent variant $[f;g]$ is pronounced \case f or g".

$$\begin{aligned}
&[:;:](A;B;C\ j\ Type) :\ (A\ !\ C)\ !\ (B\ !\ C)\ !\ (A+B)\ !\ C\\
&\qquad ::=\ \ elim^{+}{}_{\_:A+B:C}
\end{aligned}$$

Similar to the case of products, the symbol + is overloaded to also denote the bifunctor on coproducts.

$$\begin{aligned}
&:+:(A;B;C;D\ j\ Type;\ f : A\ !\ B;\ g : C\ !\ D) : (A+C)\ !\ (B+D)\\
&\qquad ::=\ [inl_{B;D} \quad f;\ inr_{B;D} \quad g]
\end{aligned}$$

Unlike products or coproducts, the datatype of parametric lists with constructors *nil* and $(:::: :)$ is an example of a genuinely recursive datatype.

$$\begin{aligned}
L\ &:\ Type\ !\ Type\\
nil\ &:\quad A : Type{:}L(A)\\
(:::: :)\ &:\quad A\ j\ Type{:}A \quad L(A)\ !\ L(A)
\end{aligned}$$

These declarations correspond to formation and introduction rules and completely determine the form of list elimination,[2]

$$\begin{aligned}
elim^{L}\ :\quad & A\ j\ Type;\ C : L(A)\ !\ Type:\\
&\quad (C(nil_A) \quad (\quad (a;l) : (A \quad L(A)){:}C(l)\ !\ C(a :: l))\\
&\qquad !\quad l : L(A){:}C(l)
\end{aligned}$$

and of the rewrites corresponding to equality rules:

$$\begin{aligned}
elim^{L}_{C}\ f\ nil_A &\rightsquigarrow fst(f)\\
elim^{L}_{C}\ f\ (a :: l) &\rightsquigarrow snd(f)\ (a;l)\ (elim^{L}_{C}\ f\ l)
\end{aligned}$$

The non-dependent variants rec^{L} and hom^{L} of list elimination are used to encode structural recursive functions on lists.

$$\begin{aligned}
&rec^{L}(A;C\ j\ Type)(f : C \quad ((A \quad L(A))\ !\ C\ !\ C))\ :\ L(A)\ !\ C\\
&\qquad ::=\ \ elim^{L}{}_{\_:L(A):C}(f)
\end{aligned}$$

$$\begin{aligned}
&hom^{L}(A;C\ j\ Type)(f : C \quad ((A \quad C)\ !\ C)) :\ L(A)\ !\ C\\
&\qquad ::=\ \ rec^{L}(fst(f);\quad (a;\_) : A \quad L(A);\ y : C{:}snd(f)(a;y))
\end{aligned}$$

[2] Bindings may also employ pattern matching on pairs; for example, a is of type A and l of type $L(A)$ in the binding $(a;l) : (A \quad L(A))$.

The name hom^L stems from the fact that $hom^L(f)$ can be characterized as a (unique) homomorphism from the algebra associated with the L datatype into an appropriate target algebra specified by f [28]. Consider, for example, the prototypical definition of the map^L functional by means of the homomorphic functional hom^L.

$$\begin{aligned} & map^L(A;B\ j\ Type)(f : A\ !\ B) :\ L(A)\ !\ L(B) \\ & \quad ::=\ hom^L(nil_B;\quad (a;y) : (A\quad L(B)).f(a) :: y\ ) \end{aligned}$$

In the rest of this paper we assume the inductive datatypes 0, 1, , $+$, $\mathbb{B}$, $\mathbb{N}$, and L together with the usual standard operators and relations on these types to be predefined.

N-ary Products and Coproducts. Using higher-order abstraction, type universes, and parametric lists it is possible to internalize n-ary versions of binary type constructors. Consider, for example, the n-ary product $A_1\quad :::\quad A_n$. It is constructed from the iterator $(:)$ applied to a list containing the types A_1 through A_n. (l) represents an n-ary coproduct constructor.

$$\begin{aligned} (:)\ :\ L(Type)\ !\ Type &\quad ::=\quad hom^L(1;\quad) \\ (:)\ :\ L(Type)\ !\ Type &\quad ::=\quad hom^L(0;\ +) \end{aligned}$$

The map^L function on lists can be used to generalize $:\ +\ :$ and $:\quad :$ to also work for the n-ary coproduct $(:)$ and the n-ary product $(:)$, respectively. Let $A\ j\ Type$, $D;R : A\ !\ Type$, $f :\quad x : A.D(a)\ !\ R(a)$, then recursion along the structure of the n-ary type constructors is used to define these generalized mapping functions.

$$\begin{aligned} & map^+(f) :\ (\quad l\ j\ L(A).\quad (map^L\ D\ l)\ !\quad (map^L\ R\ l)) \\ & \quad ::=\ elim^L\ I_0\ (\ (a;l);\ y.f(a) + y) \end{aligned}$$

$$\begin{aligned} & map\quad (f) :\ (\quad l\ j\ L(A).\quad (map^L\ D\ l)\ !\quad (map^L\ R\ l)) \\ & \quad ::=\ elim^L\ (\ \_.\ )\ (\ (a;l);\ y.f(a)\quad y) \end{aligned}$$

Although these mappings explicitly exclude tuples with independent types, they are sufficiently general for the purpose of this paper.

3 Semantic Polytypy

In this section we describe a type-theoretic framework for formalizing polytypic programs and proofs. Datatypes are modeled as initial objects in categories of functor-algebras [14], and polytypic constructions | both programs and proofs | are formulated using initiality without reference to the underlying structure of datatypes. We exemplify polytypic program construction in this type-theoretic framework with a polytypic version of the well-known *map* function. Other polytypic developments from the literature (e. g. [2]) can be added easily. Furthermore we generalize some categorical notions to also work for eliminations and lift a reflection and a fusion theorem to this generalized framework. It is, however, not our intention to provide a complete formalization of category theory in type theory; we only define certain categorical notions that are necessary to express the subsequent polytypic developments. For a more elaborated account of category theory within type theory see e. g. [8].

Functors are twofold mappings: they map source objects to target objects and they map morphisms of the source category to morphisms of the target category with the requirement that identity arrows and composition are preserved. Here, we restrict the notion of functors to the category of types (in a fixed, but sufficiently large type universe $Type_i$) with (total) functions as arrows.

Definition 1 (Functor).

$$
\begin{array}{l}
Functor : Type ::= \\
\quad F_{obj} : Type \to Type; \\
\quad F_{arr} : \Pi A;B \mid Type.\ (A \to B) \to F_{obj}(A) \to F_{obj}(B). \\
\qquad \forall A \mid Type.\ F_{arr}(I_A) \doteq I_{F_{obj}(A)} \\
\qquad \wedge\ \forall A;B;C \mid Type;\ g : A \to B;\ f : B \to C. \\
\qquad\qquad F_{arr}(f \circ g) \doteq F_{arr}(f) \circ F_{arr}(g)
\end{array}
$$

Bifunctors. Generalized functors are functors of type $\overbrace{Type \times \dots \times Type}^{n \text{ times}} \to Type$; they are used to describe datatypes with n parameter types. In order to keep the technical overhead low, however, we restrict ourselves in this paper to modeling datatypes with one parameter type only by means of bifunctors. Bifunctors are functors of type $Type \times Type \to Type$ or, using the isomorphic curried form, of type $Type \to Type \to Type$. For a bifunctor, the functor laws in Definition 1 take the following form:

Definition 2 (Bifunctor).

$$
\begin{array}{l}
Bifunctor : Type ::= \\
\quad FF_{obj} : Type \to Type \to Type; \\
\quad FF_{arr} : \Pi A;B;C;D \mid Type.(A \to B) \to (C \to D) \to \\
\qquad\qquad FF_{obj}\ A\ C \to FF_{obj}\ B\ D. \\
\qquad \forall A;B \mid Type.\ FF_{arr}\ I_A\ I_B \doteq I_{FF_{obj}\ A\ B} \\
\qquad \wedge\ \forall A;B;C;D;E;F \mid Type; \\
\qquad\quad h : A \to B;\ f : B \to C;\ k : D \to E;\ g : E \to F. \\
\qquad\qquad FF_{arr}(f \circ h)(g \circ k) \doteq (FF_{arr}\ f\ g) \circ (FF_{arr}\ h\ k)
\end{array}
$$

Many interesting examples of bifunctors are constructed from the unit type and from the product and coproduct type constructors. Seeing parameterized lists as cons-lists with constructors $nil_A : L(A)$ and $cons_A : A \times L(A) \to L(A)$ we get the bifunctor FF^L.

Example 1 (Polymorphic Lists). Let $A;B;X;Y : Type$, $f : A \to B$, $g : X \to Y$; there exists a proof term p such that:

$$FF^L : Bifunctor ::= (\lambda A;X.\,1 + (A \times X);\ \lambda f;g.\,I_1 + (f \times g);\ p)$$

Fixing the first argument in a bifunctor yields a functor.

Definition 3. *For all $(FF_{obj}; FF_{arr}; q)$: Bifunctor and A : Type there exists a proof term p such that*

$$\begin{aligned} &induced(FF_{obj}; FF_{arr}; q)(A) : \ Functor \ ::= \\ &\quad (FF_{obj}(A); \ \ X; Y \ j \ Type; \ f : X \ ! \ Y : FF_{arr} \ I_A \ f; \ p) \end{aligned}$$

Example 2. F^L : *Type* ! *Functor* ::= *induced*(FF^L)

Functor Algebras. Using the notion of *Functor* one defines the concept of datatype (see Def. 8) without being forced to introduce a signature, that is, names and typings for the individual sorts (types) and operations involved. The central notion is that of a functor algebra $Alg(F; X)$, where F is a functor. The second type definition below just introduces a name for the signature type $^C(F; T)$ of so-called catamorphisms $(j : j)$.

Definition 4. *Let F : Functor and $X; T$: Type; then:*

$$\begin{aligned} Alg(F; X) &: \ Type \ ::= \ F_{obj}(X) \ ! \ X \\ ^C(F; T) &: \ Type \ ::= \ \ X \ j \ Type{:} \, Alg(F; X) \ ! \ (T \ ! \ X) \end{aligned}$$

The initial F-algebra, denoted , is an initial object in the category of F-algebras. That is, for every F-algebra f there exists a unique object, say $(j\, f\, j)$, such that the following diagram commutes:

$$\begin{tikzcd} F(T) \arrow[r] \arrow[d, "F(j\,f\,j)"'] & T \arrow[d, "(j\,f\,j)"] \\ F(X) \arrow[r, "f"] & X \end{tikzcd}$$

In the case of lists, the initial $F^L(A)$-algebra ^{L} is defined by case split (see Section 2) and the corresponding catamorphism is a variant of the homomorphic functional on lists.

Example 3.

$$\begin{aligned} ^L(A \ j \ Type) : \ & Alg(F^L(A); L(A)) \ ::= \ [nil_A; \ cons_A] \\ (j : j)^L(A \ j \ Type) : \ & ^C(F^L(A); L(A)) \ ::= \ \ X \ j \ Type; \ f : Alg(F^L(A); X){:} \\ & \qquad hom^L(f \ \ inl_{1;A \ \ X}; \ f \ \ inr_{1;A \ \ X}) \end{aligned}$$

Paramorphisms correspond to structural recursive functions, and can be obtained by computing not only recursive results as is the case for catamorphisms but also the corresponding data structure. The definition of functor algebras and signatures for paramorphisms are a straightforward generalization of Definition 4.

Definition 5. *Let F : Functor and $X;T$: Type; then:*

$$Alg^P(F;T;X) : Type \quad ::= \quad F_{obj}(T \quad X) \ ! \ X$$
$$^P(F;T) : Type \quad ::= \quad X \ j \ Type{:}\,Alg^P(F;T;X) \ ! \ (T \ ! \ X)$$

Any F-algebra f can be lifted to the corresponding notion for paramorphisms using the function $"(:)$ in Definition 6.

Definition 6. *Let F : Functor, $T;X$: Type; then:*

$$"(f : Alg(F;X)) : Alg^P(F;T;X) \quad ::= \quad f \quad F_{arr}(\ ^2_{T;X})$$

It is well-known that the notions of catamorphisms and paramorphisms are interchangeable, since one can define a paramorphism $[\![:]\!]$ from a catamorphism $(\!j:j\!)$ and vice versa.

$$(\!j:j\!) : \quad ^C(F;T) \quad ::= \quad X \ j \ Type{:}[\![:]\!] \quad "(:)$$
$$[\![:]\!] : \quad ^P(F;T) \quad ::= \quad X \ j \ Type;\ g : Alg^P(F;T;X);\ t : T{:}$$
$$^2((\!j \quad z : F(T \quad X){:}(t;\ g(z))\,j\!)(t))$$

Eliminations, however, are a genuine generalization of both catamorphisms and paramorphisms in that they permit defining functions of dependent type. In this case, the notion of functor algebras generalizes to a type $Alg^E(F;C;\ )$ that depends on an F-algebra , and the signature type $^E(F;T;\ )$ for eliminations is expressed in terms of this generalized notion of functor algebra.

Definition 7. *Let F : Functor, T : Type, $C : T \ ! \ Type$, $: Alg(F;T)$; then:*

$$Alg^E(F;C;\ ) : Type \quad ::= \quad z : F_{obj}(\ x : T{:}C(x)){:}C((\quad F_{arr}(\ ^1_{T;C}))\ z)$$
$$^E(F;T;\ ) : Type \quad ::= \quad C : T \ ! \ Type{:}\,Alg^E(F;C;\ ) \ ! \quad x : T{:}C(x)$$

These definitions simplify to the corresponding notions for paramorphisms (see Def. 5) when instantiating C with a non-dependent type of the form $(\ \_ : T{:}X)$. Moreover, the correspondence between the induction hypotheses of the intuitive structural induction rule and the generalized functor algebras $Alg^E(F^L;C;\ ^L)$ is demonstrated in [19].

The definition of $_C$ below is the key to using the usual categorical notions like initiality, since it transforms a generalized functor algebra f of type $Alg^E(F;C;\ )$ to a functor algebra $Alg(F;\ x : T{:}C(x))$.[3]

Definition 8. *Let F : Functor, T : Type, $: Alg(F;T)$, $elim : \ ^E(F;C;\ )$, $C : T \ ! \ Type$, and $f : Alg^E(F;C;\ )$; then:*

$$_C(f) : Alg(F;\ x : T{:}C(x)) \quad ::= \quad h \quad F_{arr}(\ ^1_{T;C});\ f\,i$$

[3] Here and in the following the split function $h{:};{:}i$ is assumed to also work for function arguments of dependent types; i.e.$hf;gi : \ x : B{:}C(x) \ ::= \ (f(x);g(x))$ where $A;B$: Type, $C : B \ ! \ Type$, $f : A \ ! \ B$, $g : \ x : A{:}C(f(x))$, and $x : A$.

$$\begin{tikzcd}
F(T) \arrow[rr] \arrow[d, "F(R_C(f))"'] & & T \arrow[d, "R_C(f)"] \\
F(\ x : T{:}C(x)) \arrow[rr, "{}_C(f)"] & & x : T{:}C(x)
\end{tikzcd}$$

Fig. 1. Universal Property.

Now, we get the initial algebra diagram in Figure 1 [19], where $R_C(f)$ denotes the unique function which makes this diagram commute. It is evident that the first component must be the identity. Thus, $R_C(f)$ is of the form $hI;\ elim_C(f)i$ where the term $elim_C(f)$ is used to denote the unique function which makes the diagram commute for the given f of type $Alg^E(F;C;\ )$. Altogether, these considerations motivate the following universal property for describing initiality.

Definition 9.

$$\begin{aligned}
&universal^E(F;T;\ ) : \ Prop \quad ::= \\
&\qquad C : T \ \ ! \ \ Type;\ f : Alg^E(F;C;\ ){:}9^1\, E :_{.} (\ \ x : T{:}C(x)){:} \\
&\qquad\quad \mathrm{let}\ R_C(f) \ = \ hI;\, E\, i \ \mathrm{in}\ R_C(f) \quad \doteq \ {}_C(f) \ \ F(R_C(f))
\end{aligned}$$

Witnesses of this existential formula are denoted by $elim_C(f)$ *and* $R_C(f)$ *is defined by* $hI;\ elim_C(f)i$.

Reflection and Fusion. Eliminations enjoy many nice properties. Here, we concentrate on some illustrative laws like reflection or fusion, but other laws for catamorphisms as described in the literature can be lifted similarly to the case of eliminations.

Lemma 1 (Reflection). *Let* $universal^E(F;T;\ )$ *be inhabited; then:*

$$R_{\_:T:T}(\,''(\ \ ))\quad {}^2_{T;T} \ \dot{\doteq} \ \ x : T{:}(x;x)$$

This equality follows directly from uniqueness and some equality reasoning.

The fusion law for catamorphism is central for many program manipulations [2]. Now, this fusion law is generalized to also work for eliminations.

Lemma 2 (Fusion). *For functor* F *and type* T*, let* $C;D : T \ \ ! \ \ Type$*,* $f : Alg^E(F;C;\ )$*,* $g : Alg^E(F;D;\ )$*,* $h : \ \ x : T{:}C(x) \ \ ! \ \ \ x : T{:}D(x)$*, and assume that the following holds:*

$$\begin{aligned}
&H_1 : universal^E(F;T;\ ) \\
&H_2 : \ \ S : Type; E : S \ \ ! \ \ Type; \underset{.}{u}; v : Alg^E(F;E;\ ){:} \\
&\qquad\qquad u \ \doteq \ v \ \ ) \ \ elim_E(u) \doteq elim_E(v)
\end{aligned}$$

Then: $h \quad {}_C(f) \ \dot{\doteq} \ \ {}_D(g) \ \ F(h) \quad) \quad h \ \ R_C(f) \ \dot{\doteq} \ R_D(g)$

The proof of this generalized fusion theorem is along the lines of the fusion theorem for catamorphisms [2].

$$\begin{tikzcd}
F(T) \arrow[r, "F(R_C(f))"] \arrow[d] & F(\quad x:T{:}C(x)) \arrow[r, "F(h)"] \arrow[d, "{}_C(f)"] & F(\quad x:T{:}D(x)) \arrow[d, "{}_D(g)"] \\
T \arrow[r, "R_C(f)"'] & x:T{:}C(x) \arrow[r, "h"'] & x:T{:}D(x)
\end{tikzcd}$$

This diagram commutes because the left part does (by definition of elimination) and the right part does (by assumption). (Extensional) equality of $h \quad R_C(f)$ and $R_D(g)$ follows from the uniqueness part of the universality property and the functoriality of F. The extra hypothesis H_2 is needed for the fact that the binary relation $\doteq$ (see Section 2) on functions is not a congruence in general.

Catamorphisms and Paramorphisms. A paramorphism is simply a non-dependent version of an elimination scheme and a catamorphism is defined in the usual way from a paramorphism.

Definition 10. *Let universal$^E(F;T;\quad)$ be inhabited and X : Type; then:*

$$\begin{array}{llll}
[\!(\,.\,)\!] : & {}^P(F;T) & ::= & X\ j\ Type{:}elim_{(\ \_:T:X)} \\
(\!|\,.\,|\!) : & {}^C(F;T) & ::= & X\ j\ Type{:}[\!(\,.\,)\!]\quad "\,(.)
\end{array}$$

Lemma 3. *If f : $Alg(F;X)$, g : $Alg^P(F;T;X)$, and universal$^E(F;T;\quad)$ is inhabited then $[\!(\,.\,)\!]$ and $(\!|\,.\,|\!)$ are the unique functions satisfying the equations:*

$$\begin{array}{lll}
(\!| f |\!) & \doteq & f \quad F((\!| f |\!)) \\
[\!(g)\!] & \doteq & g \quad F(hI_T;[\!(g)\!]i)
\end{array}$$

Example 4. Let FF : *Bifunctor* then the polytypic *map* function is defined by

$$map(f) : T(A)\ !\ T(B) \ ::= \ (\!| \qquad F(f)(I_{T(B)})\,|\!)$$

Uniform Specification of Datatypes in ECC. Now, the previous developments are used to specify, in *ECC*, classes of parametric datatypes in a uniform way. The exact extension of this class is unspecified, and we only assume that there is a name *dt* and a describing bifunctor FF_{dt} for each datatype in this class.

Definition 11 (Specifying Datatypes). *For name : Type, let C : $T\ !\ Type$, and FF : name ! Bifunctor.*
Define $F(dt : name)$: Type ! functor $::=$ induced(FF_{dt}); then:

Formation: T : $name\ !\ Type\ !\ Type$

Introduction: $\quad$: $dt\ j\ name;\ A\ j\ Type{:}\ Alg(F_{dt}(A);T_{dt}(A))$

Elimination: $elim$: $dt\ j\ name;\ A\ j\ Type{:}\quad {}^E(F_{dt}(A);T_{dt}(A);\ {}_{dt;A})$

Equality:

$$\begin{array}{lll}
(F_{dt}(A)(\ {}^1_{T;C})\quad F_{dt}(A)(R_C(f))) & \rightsquigarrow & I_{F_{dt}(A)(T_{dt}(A))} \\
elim_C(f)(\ \ (x)) & \rightsquigarrow & f(F_{dt}(A)\ R_C\ x)
\end{array}$$

It is straightforward to declare three constants T, , *elim* corresponding to formation, introduction, and elimination rule, respectively. The equality induced by the diagram in Figure 1 is $elim_C(f) \quad \doteq f \quad F(R_C)$. It gives the following reduction rule by expressing the equality on the point level and by replacing equality by reducibility ($x : F(T)$): $elim_C(f)(\;(x)) \rightsquigarrow f(F\;\;R_C\;\;x)$. There is a slight complication, however, since the left-hand side of this reduction rule is of type $C(\;(x))$ while the right-hand side is of type $C(\;((F(\;^{1}_{T;C})\;\;F(R_C(f)))\,x)$. These two types, although provably Leibniz-equal, are not convertible. Consequently, an additional reduction rule, which is justified by the functoriality of F, is needed. This extraneous reduction rule is not mentioned for the extension of *ECC* with datatypes in [19], since the implicit assumption is that a reduction rule is being added for each functor of a syntactically closed class of functors. In this case, the types of the left-hand and right-hand sides are convertible for each instance. Since we are interested in specifying datatypes uniformly, however, we are forced to abstract over the class of all functors, thereby loosing convertibility between these types.

4 Syntactic Polytypy

The essence of polytypic abstraction is that the syntactic structure of a datatype completely determines many developments on this datatype. Hence, we specify a syntactic representation for making the internal structure of datatypes explicit, and generate, in a uniform way, bifunctors from datatype representations. Proofs for establishing the bifunctoriality condition or the unique extension property follow certain patterns. The order of applying product and coproduct inductions in the proof of the existential direction of the unique extension property, for example, is completely determined by the structure of the underlying bifunctor. Hence, one may develop specialized tactics that generate according proofs separately for each datatype under consideration. Here, we go one step further by capturing the general patterns of these proofs and internalizing them in type theory. In this way, polytypic proofs of the applicability conditions of the theory formalized in this section are constructed *once and for all*, and the proof terms for each specific datatypes are obtained by simple instantiation.

These developments are used to instantiate the abstract parameters *name* and *FF* in order to specify a syntactically characterized class of datatypes. Fixing the shape of datatypes permits defining polytypic constructions by recursing (inducting) along the structure of representations. The additional expressiveness is demonstrated by means of defining polytypic recognizers and a polytypic *no confusion* theorem.

In order to keep subsequent developments manageable and to concentrate on the underlying techniques we choose to restrict ourselves to representing the | rather small | class of parametric, polynomial datatypes; it is straightforward, however, to extend these developments to much larger classes of datatypes like the ones described by (strictly) positive functors [19, 5]. The only restriction on the choice of datatypes is that the resulting reduction relation as specified in

Figure 11 is strongly normalizing. The developments in this section are assumed to be implicitly parameterized over the entities of Figure 11.

A natural representation for polynomial datatypes is given by a list of lists, whereby the j^{th} element in the i^{th} element list determines the type of the j^{th} selector of the i^{th} constructor. The type *Rep* below is used to represent datatypes with n constructors, where n is the length of the representation list, and the type *Sel* restricts the arguments of datatype constructors to the datatype itself (at recursive positions) and to the polymorphic type (at non-recursive positions), respectively. Finally, *rec* and *nonrec* are used as suggestive names for the injection functions of the *Kind* coproduct.

Definition 12 (Representation Types).

$$\begin{aligned} Kind : Type &\quad ::= \quad rec : 1 + nonrec : 1 \\ Sel : Type &\quad ::= \quad L(Kind) \\ Rep : Type &\quad ::= \quad L(Sel) \end{aligned}$$

Consider, for example, the representation dt^B for binary trees below. The lists *nil* and ($nonrec :: rec :: rec :: nil$) describe the signatures of the tree constructors *leaf* and *node*, respectively.

Example 5. $dt^B \; : \; Rep \quad ::= \quad nil :: (nonrec :: rec :: rec :: nil) :: nil$

The definitions below introduce, for example, suggestive names for the formation type and constructors corresponding to the representation dt^B in Example 5.

Example 6. Let $B : Type \;\; ! \;\; Type \quad ::= \quad T(dt^B)$; then

$$leaf(A : Type) : \;\; 1 \;\; ! \;\; B(A) \quad ::= \quad (\;\; {}_{dt^B;A} \quad inl_{1;A+B(A)+B(A)+0})$$

$$\begin{aligned} &node(A \; j \; Type) : \;\; (A \quad B(A) \quad B(A) \quad 1) \;\; ! \;\; B(A) \quad ::= \\ &\qquad (\;\; {}_{dt^B;A} \quad inr_{1;A+B(A)+B(A)+0} \quad inl_{A+B(A)+B(A);0}) \end{aligned}$$

Argument types $Arg_{A;X}(s)$ of constructors corresponding to the selector representation s are computed by placing the (parametric) type A at non-recursive positions, type X at recursive positions, and by forming the n-ary product of the resulting list of types.

Definition 13.

$$Arg(A; X \; : Type)(s : Sel) : \;\; Type \quad ::= \qquad (map^L \;\; (rec^+ \;\; X \;\; A) \;\; s)$$

Next, a polytypic bifunctor *FF* is computed uniformly for the class of representable datatypes. The object part of these functors is easily computed by forming the n-ary sum of the list of argument types (products) of constructors. Likewise the arrow part of *FF* is computed by recursing over the structure of the representation type *Rep*. This time, however, the recursion is a bit more involved, since all the types of resulting intermediate functions depend on the form of the part of the representation which has already been processed.

Proposition 1. *For all dt : Rep there exists a proof object p such that*

$$
\begin{aligned}
&FF(dt) : Bifunctor &&::= (FF^{dt}_{obj}, FF^{dt}_{arr}, p)\\
&\text{where } FF^{dt}_{obj} &&::= (\ \ A, X : Type.\ \ ()\ \ map^{L}(Arg_{A,X}))\ dt\\
&\qquad FF^{dt}_{arr} &&::= \ \ A, B, X, Y \mid Type, f : A \to B, g : X \to Y.\\
& && \qquad (map^{+}_{Arg_{A,X},\ Arg_{B,Y}}\ \ map_{(rec^{+}\ A\ X),\ (rec^{+}\ B\ Y)})\\
& && \qquad\quad (elim^{+}_{(\ \ k:Kind.(rec^{+}\ A\ X\ k) \to (rec^{+} B\ Y\ k))}\ f\ g)
\end{aligned}
$$

The inductive construction of the bifunctoriality proof p parallels the structure of the recursive definition of $FF(dt)$. We present one part of this proof — the preservation of identities — in more detail. Let $dt : Rep$, and $A, X : Type$. The goal is to show that $FF^{dt}_{arr}\ I_A\ I_X \doteq I_{FF^{dt}_{obj}\ A\ X}$. The proof is by induction on the representation type dt. The base case where $dt = nil$ represents an empty datatype and hence the proposition is trivially true. In the induction step one has to prove that $\forall x : FF^{s::l}_{obj}.FF^{s::l}_{arr}\ I_A\ I_X\ x = I_{FF^{s::l}_{obj}\ A\ X}\ x$ given that the proposition holds for l. The left-hand side of this equation evaluates to

$$
\begin{aligned}
&(map\ \ (elim^{+}_{C}\ I_A\ I_X) + FF^{l}_{arr}\ I_A\ I_X)\ x\\
&\text{where } C ::= \ \ k : Kind.(rec^{+}\ A\ X\ k) \to (rec^{+} A\ X\ k)
\end{aligned}
$$

We proceed by a coproduct induction on x. The *inr* case can be proved easily using the induction hypothesis. For the *inl* case we have to show that

$$\forall y : Arg_{A,X}(s).map\ \ (elim^{+}_{C}\ I_A\ I_X)\ y = I_{Arg_{A,X}(s)}\ y$$

The next step is to induct on the representation s of the argument type of a constructor. The base case corresponds to a zero-place constructor and holds trivially. The induction step requires us to prove that

$$\forall y : Arg_{A,X}(k :: l^0).map_{k::l'}(elim^{+}_{C}\ I_A\ I_X)\ y = I_{Arg_{A,X}(k::l')}\ y$$

under the induction hypothesis that the proposition holds for l^0. The left-hand side of this equation evaluates to $(elim^{+}_{C}\ I_A\ I_X\ k\ \ \ map_{l'}(elim^{+}_{C}\ I_A\ I_X))\ y$. The induction hypothesis reduces the right-hand side function to $I_{Arg_{A,X}(l')}$ and by simple case analysis on k one can prove that $elim^{+}_{C}\ I_A\ I_X\ k = I_{rec^{+}\ A\ X\ k}$. Thus, the goal now reads $(I_{rec^{+}\ A\ X\ k}\ \ \ I_{Arg_{A,X}(l')})\ y = I_{Arg_{A,X}(k::l')}\ y$. This reduces to the trivial goal $(fst(y), snd(y)) = y$ This polytypic proof has been constructed in Lego using 14 refinement steps.

The second part of the bifunctoriality proof, the preservation of composition, runs along the same lines. More precisely, the induction proceeds by inducting on the number of coproduct inductions $elim^{+}$ as determined by the length of the representation type dt followed by an induction on the number of product inductions *elim* in the induction step; the outer (inner) induction employs one coproduct (product) induction $elim^{+}$ (*elim*) in its induction step.

n-th Constructor. Example 6 suggests encoding a function for extracting the n-th constructor from $_{dt}$. Informally, this function chains n right-injections with a final left-injection: $c(n) \quad {}_{dt;A} \quad inr \;\; ::: \;\; inr \quad inl$. It is clear how to internalize the dots by recursing on n, but the complete definition of this function is somewhat involved for the dependency of the right injections from the position i and the types of intermediate functions (see also Example 6). Thus, we restrict ourselves to only state the type of this function.

$$
\begin{aligned}
c: \quad & dt \; j \; Rep; \;\; A \; j \; Type; \;\; n : \mathbb{N}; \;\; p : (n < len(dt)): \\
& \quad Arg_{A;T_{dt}(A)} \; (nth \; dt \; n \; p) \;\; ! \;\; T_{dt}(A)
\end{aligned}
$$

Polytypic Recognizer. The explicit representation of datatypes permits defining a function for computing recognizer functions for all (representable) datatypes. First, the auxiliary function r traverses a given representation list and returns a pair $(f;i)$ consisting of a recognizer f and the current position i in the representation list.

Definition 14. *Let $dt : Rep$, $A : Type$, $n : \mathbb{N}$, and $p : (n < len(dt))$.*

$$
\begin{aligned}
rcg(dt;A;n;p) \quad &: \quad Alg(F_{dt}(A);Prop) \quad ::= \quad {}^{1}(r^{dt}_{A;n;p}) \\
r(dt;A;n;p) \quad &: \quad Alg(F_{dt}(A);Prop) \quad\; \mathbb{N} \\
r^{nil}_{A;n;p} \quad &= \quad (\;\; x : 0 \text{:}arbitrary_{Prop} \; x; \;\; zero) \\
r^{s::l}_{A;n;p} \quad &= \quad \text{let } (f;i) \; = \; r^{l}_{A;n;p} \text{ in} \\
& \qquad\qquad ([\;\; \_ : Arg_{A;Prop}(s)\text{:}n \; = \; i; \;\; f]; \;\; succ(i))
\end{aligned}
$$

Now, it is a simple matter to define the polytypic recognizer by applying the polytypic catamorphism on *rcg*.

Definition 15 (Polytypic Recognizer). *Let $dt : Rep$, $A : Type$, $n : \mathbb{N}$, and $p : (n < len(dt))$.*

$$
R(dt;A;n;p) : \;\; T_{dt}(A) \;\; ! \;\; Prop \quad ::= \quad (\!j \; rcg^{dt}_{A;n;p} \; j\!)
$$

This function satisfies the following characteristic properties for recognizers.

Proposition 2. *Let $dt \; j \; Rep$, $A \; j \; Type$, $i;j : \mathbb{N}$; furthermore $p : (i < len(dt))$, $q : (j < len(dt))$, $a : Arg_{A;T_{dt}(A)}(nth \; dt \; i \; p)$, then:*

1. $R_{i;p} \;\; c_{i;p}(a)$
2. $i \not= j \;\;) \;\; : \; (R_{j;q} \;\; c_{i;p}(a))$

Polytypic No Confusion. Now, we have collected all the ingredients for stating and proving a polytypic *no confusion* theorem once and for all.

Theorem 1 (Polytypic No Confusion).

$$
\begin{aligned}
& dt \; j \; Rep; \;\; A \; j \; Type; \;\; i;j : \mathbb{N}; \;\; p : (i < len(dt)); \;\; q : (j < len(dt)); \\
& a : \;\; Arg_{A;T_{dt}(A)} \; (nth \; dt \; i \; p); \;\; b : \;\; Arg_{A;T_{dt}(A)} \; (nth \; dt \; j \; q) \\
& \qquad i \not= j \;\;) \;\; c_{i;p}(a) \not= c_{j;q}(b)
\end{aligned}
$$

Given the hypothesis $H : c_{i;p}(a) = c_{j;q}(b)$ one has to construct a proof term of $?$. According to Proposition 2 this task can be reduced to finding a proof term for $R_{j;q}\ c_{i;p}(a)$. Furthermore, for hypothesis H, this goal is equivalent with formula $R_{j;q}\ c_{j;q}(a)$, which is trivially inhabited according to Proposition 2. This finishes the proof. Again, each of these steps correspond to a refinement step in the Lego formalization. Notice that the proof of the polytypic bifunctoriality property and the polytypic *no confusion* theorem are as straightforward as any instance thereof for any datatype and for any legitimate pair of datatype constructors.

5 Conclusions

The main conclusion of this paper is that the expressiveness of type theory permits internalizing many interesting polytypic constructions that are sometimes thought to be external and not formalizable in the base calculus [22, 24]. In this way, polytypic abstraction in type theory has the potential to add another level of flexibility in the reusability of formal constructions and in the design of libraries for program and proof development systems. We have demonstrated the feasibility of our approach using some small-sized polytypic constructions, but, obviously, much more work needs to be done to build a useful library using this approach. Most importantly, a number of polytypic theorems and polytypic proofs thereof need to be identified. Although we have used a specific calculus, namely the Extended Calculus of Constructions, for our encodings of *semantically* and *syntactically* polytypic abstraction, similar developments are possible for other type theories such as Martin-Löf type theories [18] with universes or the Inductive Calculus of Constructions.

Semantically polytypic developments are formulated using initiality without reference to the underlying structure of datatypes. We have demonstrated how to generalize some theorems from the literature, like reflection and fusion for catamorphisms, to corresponding theorems for dependent paramorphisms (eliminations). These developments may not only make many program optimizations, like the fusion theorem above, applicable to functions of dependent type but | for the correspondence of dependent paramorphisms with structural induction | it may also be interesting to investigate usage of these generalized polytypic theorems in the proof development process; consider, as a simple example, a polytypic construction of a double induction scheme from structural induction.

Syntactic polytypism is obtained by syntactically characterizing the descriptions of a class of datatypes. Dybjer and Setzer [5] extend the notion of inductively defined sets (datatypes) that are characterized by strictly positive functors [19, 21] and provide an axiomatization of sets defined through induction-recursion. For the purpose of demonstration, however, we have chosen to deal with a special case of datatypes | the class of polynomial datatypes | in this paper, but it is straightforward, albeit somewhat more tedious, to generalize these developments to support larger classes. From a technical point of view, it was essential to be able to recurse along the structure of arbitrary product types in order to encode in type theory many $: : :$ as used in informal developments. We solved this problem by describing product types A_1 ::: A_n by a list l of the types A_i and by encoding a function (l) for computing the corresponding product type. These developments may also be interesting for other applications such as statically typing heterogeneous lists (or S-expressions) within type theory. The first step towards syntactic polytypism consists in fixing a representation type for the chosen class of datatypes; this representation (or abstract syntax) can be understood as a simple kind of *meta-level* representation, since it makes the internal structure of datatype descriptions amenable for inspection. In the next step, one fixes the denotation of each datatype representation by assigning a corresponding functor to it. These functor terms can be thought of as being on the *object-level* and the type-theoretic function for computing denotations of representation is sometimes called a *computational reflection* function. This internalization of both the representation and the denotation function permits abstracting theorems, proofs, and programs with respect to the class of (syntactically) representable datatypes.

The added expressiveness of syntactic polytypy has been demonstrated by means of several examples. The polytypic (bi)functoriality proof e. g. requires inducting on syntactic representations, and the proof of the *no confusion* theorem relies on the definition of a family of polytypic recognizer functions. This latter theorem is a particularly good example of polytypic abstraction, since its polytypic proof succinctly captures a 'proof idea' for an infinite number of datatype instances.

References

[1] B. Barras, S. Boutin, and C. Cornes et. al. *The Coq Proof Assistant Reference Manual - Vers. 6.2.4.* INRIA, Rocquencourt, 1998.

[2] R. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.

[3] C. Böhm and A. Berarducci. Automatic Synthesis of Typed -Programs on Term Algebras. *Theoretical Computer Science,* 39:135{154, 1985.

[4] T. Coquand and C. Paulin. Inductively Defined Types. In *Proc. COLOG 88,* volume 417 of *LNCS,* pages 50{66. Springer-Verlag, 1990.

[5] P. Dybjer and A. Setzer. A Finite Axiomatization of Inductive-Recursive Definitions. In J.-Y. Girard, editor, *Proc. 4th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'99,* volume 1581 of *LNCS.* Springer-Verlag, 1999.

[6] M. J. C. Gordon and T. F. Melham (eds.). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[7] R. Harper and R. Pollack. Type Checking, Universal Polymorphism, and Type Ambiguity in the Calculus of Constructions. In *TAPSOFT'89, volume II,* LNCS, pages 240{256. Springer-Verlag, 1989.

[8] G. Huet and A. Sa¨bi. Constructive Category Theory. In *Proc. CLICS-TYPES Workshop on Categories and Type Theory,* January 1995.

[9] C.B. Jay and J.R.B. Cockett. Shapely Types and Shape Polymorphism. In D. Sannella, editor, *Programming Languages and Systems { ESOP'94,* volume 788 of *LNCS,* pages 302{316. Springer-Verlag, 1994.

[10] J. Jeuring. Polytypic Pattern Matching. In *Conf. Functional Programming Languages and Computer Architecture (FPCA '95),* pages 238{248. ACM Press, 1995.

[11] J. Jeuring and P. Jansson. Polytypic Programming. In T. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, LNCS, pages 68{114. Springer-Verlag, 1996.

[12] Z. Luo. An Extended Calculus of Constructions. Technical Report CST-65-90, University of Edinburgh, July 1990.

[13] Z. Luo and R. Pollack. The Lego Proof Development System: A User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[14] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255{279, 1990.

[15] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413{425, 1992.

[16] L. Meertens. Calculate Polytypically. In H. Kuchen and S.D. Swierstra, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP'96)*, LNCS, pages 1{16. Springer-Verlag, 1996.

[17] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. 5th Conf. on Functional Programming Languages and Computer Architecture*, pages 124{144, 1991.

[18] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory*. Monographs on Computer Science. Oxford Science Publications, 1990.

[19] Ch.E. Ore. The Extended Calculus of Constructions (ECC) with Inductive Types. *Information and Computation*, 99, Nr. 2:231{264, 1992.

[20] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107{125, 1995.

[21] C. Paulin-Mohring. Inductive Definitions in the System Coq, Rules and Properties. In J.F. Groote M.Bezem, editor, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328{345. Springer-Verlag, 1993.

[22] L.C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definition. Technical report, Computer Laboratory, University of Cambridge, 1996.

[23] H. Rue . Computational Reflection in the Calculus of Constructions and Its Application to Theorem Proving. In J. R. Hindley P. de Groote, editor, *Proc. 3rd Int. Conf. on Typed Lambda Calculi and Applications TLCA'97*, volume 1210 of *LNCS*, pages 319{335. Springer-Verlag, 1997.

[24] N. Shankar. Steps Towards Mechanizing Program Transformations Using PVS. Preprint submitted to Elsevier Science, 1996.

[25] T. Sheard. Type Parametric Programming. Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.

[26] The Lego team. The Lego Library. Distributed with Lego System, 1998.

[27] D. Tuijnman. *A Categorical Approach to Functional Programming*. PhD thesis, Universität Ulm, 1996.

[28] F. W. von Henke. An Algebraic Approach to Data Types, Program Verification, and Program Synthesis. In *Mathematical Foundations of Computer Science, Proceedings*, volume 45 of *LNCS*. Springer-Verlag, 1976.

[29] F. W. von Henke, S. Pfab, H. Pfeifer, and H. Rue . Case Studies in Meta-Level Theorem Proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 461{478. Springer-Verlag, September 1998.

Recursive Function Definition over Coinductive Types

John Matthews

Oregon Graduate Institute,
P.O. Box 91000, Portland OR 97291-1000, USA
johnm@cse.ogi.edu
http://www.cse.ogi.edu/~johnm

Abstract. Using the notions of *unique fixed point*, *converging equivalence relation*, and *contracting function*, we generalize the technique of well-founded recursion. We are able to define functions in the Isabelle theorem prover that recursively call themselves an infinite number of times. In particular, we can easily define recursive functions that operate over coinductively-defined types, such as infinite lists. Previously in Isabelle such functions could only be defined corecursively, or had to operate over types containing “extra” bottom-elements. We conclude the paper by showing that the functions for filtering and flattening infinite lists have simple recursive definitions.

1 Well-Founded Recursion

Rather than specify recursive functions by possibly inconsistent axioms, several higher order logic (HOL) theorem provers[3, 9, 12] provide well-founded recursive function definition packages, where new functions can be defined conservatively. Recursive functions are defined by giving a series of pattern matching reduction rules, and a well-founded relation.

For example, the *map* function applies a function f pointwise to each element of a finite list. This function can be defined using well-founded recursion:

$$map :: (\alpha \to \beta) \to \alpha\ list \to \beta\ list$$

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ (x\#xs) &= (f\ x)\ \#\ (map\ f\ xs) \end{aligned}$$

The first rule states that *map* applied to the empty list, denoted by [], is equal to the empty list. The second rule states that *map* applied to a list constructed out of the head element x and tail list xs, denoted by $x\#xs$, is equal to the list formed by applying f to x and $map\ f$ to xs recursively.

To define a function using well-founded recursion, the user must also supply a *well-founded relation* on one of the function's arguments[1]. A well-founded

[1] Some well-founded recursion packages only allow single-argument functions to be defined. In this case one can gain the effect of multi-argument curried functions by tupling.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 73–90, 1999.

relation ($<$) is a relation with the property that there exists no infinite sequence of elements $x_1;x_2;x_3;x_4;:::$ such that

$$:::<x_4<x_3<x_2<x_1$$

For each reduction rule, the recursive definition package checks that every recursive call on the right-hand side of the rule is applied to a smaller argument than on the left-hand side, according to the user supplied well-founded relation.

In the case of *map*, we can supply the well-founded relation

$$xs < ys \quad length\,xs < length\,ys$$

which is true when the number of elements in the relation's left-hand list argument is less than the number of elements in the relation's right-hand argument. The definition of *map* contains only one recursive rule, and it is easy to prove that the *xs* argument of the recursive call of *map* is smaller than the $(x\#xs)$ argument on the left-hand side of the rule, according to this relation. In general, well-founded relations ensure that there are no infinite chains of nested recursive calls.

2 Coinductive Types and Corecursive Functions

Although well-founded recursion is a useful definition technique, there are many recursive definitions that fall outside its scope. For instance, there is a non-inductive type of *lazy lists* in the Isabelle[9] theorem prover, denoted by *llist*, that is the set of all finite and infinite lists of type . The function *lmap* over this type is uniquely specified by the following recursive equations[2]:

$$\begin{aligned} lmap\ f\ [] &= [] \\ lmap\ f\ (x\#xs) &= (f\ x)\ \#\ (lmap\ f\ xs) \end{aligned}$$

One cannot define *lmap* using well-founded recursion since the length of an infinite list does not decrease when you take its tail. In fact, the expression $lmap\ f\ (x_1\ \#\ x_2\ \#\ x_3\ \#\ :::)$ can be unfolded using the above rules to an infinite chain of recursive calls:

$$\begin{aligned} & lmap\ f\ (x_1\ \#\ x_2\ \#\ x_3\ \#\ :::) \\ = \\ & (f\ x_1)\ \#\ (lmap\ f\ (x_2\ \#\ x_3\ \#\ :::)) \\ = \\ & (f\ x_1)\ \#\ (f\ x_2)\ \#\ (lmap\ f\ (x_3\ \#\ :::)) \\ = \\ & (f\ x_1)\ \#\ (f\ x_2)\ \#\ (f\ x_3)\ \#\ (lmap\ f\ (:::)) \\ = \\ & \ldots \end{aligned}$$

[2] Isabelle uses a different syntax for lazy lists than for finite lists. In this paper we use the same syntax for both types.

Defining Functions Corecursively

The α *llist* type is an example of a coinductive type. Although there is no general induction principle for coinductive types, one can use principles of coinduction to show that two coinductive values are equal, and one can build coinductive values using *corecursion*.

In Isabelle's theory of lazy lists[10], for instance, one builds potentially infinite lists through the *llist_corec* operator, which has type $\beta \Rightarrow (\beta \Rightarrow unit + (\alpha \times \beta)) \Rightarrow (\alpha\ llist)$. The *llist_corec* operator uniquely satisfies the following recursion equation:

$$llist\_corec\ b\ g = \begin{cases} [], & \text{if } g\ b = \mathrm{Inl}\,() \\ (x\ \#\ (llist\_corec\ b'\ g)), & \text{if } g\ b = \mathrm{Inr}\,(x, b') \end{cases}$$

The *llist_corec* operator takes as arguments an initial value b and a function g. When g is applied to b, it either returns Inl (), indicating that the result list should be empty, or the value Inr (x, b'), where x represents the first element of the result list, and b' represents the new initial value to build the rest of the list from. Function g is called iteratively in this fashion, constructing a potentially infinite list.

Using *llist_corec*, we can define *lmap* corecursively as follows:

$$\begin{array}{l} lmap\ f\ xs \equiv llist\_corec\ xs\ (map\_head\ f) \\ \text{where} \\ \quad map\_head :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ llist \Rightarrow (unit + (\beta \times \alpha\ llist)) \\ \quad map\_head\ f\ xs \equiv \text{case}\ xs\ \text{of} \\ \qquad\qquad [] \Rightarrow \mathrm{Inl}\,() \\ \qquad\quad \mid (x \# xs') \Rightarrow \mathrm{Inr}\,(f\ x, xs') \end{array}$$

One can then prove by coinduction that this definition satisfies *lmap*'s recursive equations. Needless to say, this is not the most intuitive specification of *lmap*, and most people would prefer to specify such functions using recursion, if possible. In the remainder of the paper we will present a framework for defining functions such as *lmap* recursively.

3 Solving Recursive Equations

The basic steps required in this framework to show that a set of recursive equations is well defined are as follows:

- Express the recursive equations as a fixed point of a functional F.
- Show that for any two different potential solutions supplied to F, F maps them to two potential solutions that are closer together, in a suitable sense.
- Invoke the main result (Sect. 4.3) to show that the above property of F is sufficient to guarantee that there is a unique solution to the original set of recursive equations.

In this section we deal with the first step.

3.1 Unique Fixed Points

We convert a system of pattern matching recursive equations into a functional form by employing a standard technique from domain theory[4, 15]. We start by recasting the equations as a single recursive equation using argument destructors or nested case-expressions. For example, the recursive equations defining the *lmap* function are equivalent to the following single recursive equation:

$$\begin{array}{l} lmap\ f\ l = \text{case } l \text{ of} \\ \qquad\quad [\,] \qquad\quad \Rightarrow [\,] \\ \qquad\ \mid (x \# xs) \ \ \Rightarrow (f\ x) \# (lmap\ f\ xs) \end{array}$$

Given f, we can reify this pattern of recursion into a non-recursive functional F of type $(\alpha\ llist \to \beta\ llist) \to (\alpha\ llist \to \beta\ llist)$ that takes a function parameter *lmap_f*:

$$\begin{array}{l} F\ lmap\_f = \lambda l.\ \text{case } l \text{ of} \\ \qquad\qquad\qquad [\,] \qquad\quad \Rightarrow [\,] \\ \qquad\qquad\quad \mid (x \# xs) \ \ \Rightarrow (f\ x) \# (lmap\_f\ xs). \end{array}$$

Using the recursive equations for *lmap*, it is easy to show that $lmap\ f = F(lmap\ f)$. The value $lmap\ f$ is called a *fixed point* of F. In general, an element x of type α is a fixed point of a function g of type $\alpha \to \alpha$ if $x = g\,x$. A function may have many fixed points, or none at all. Considering g as a functional representation of a system of recursive equations, each fixed point of g represents a valid solution to the system. If the function g has exactly one fixed point x, then we can think of g as *defining* the value x. We use Hilbert's description operator (ε) to formalize this notion in HOL:

$$\begin{array}{l} \mu :: (\alpha \to \alpha) \to \alpha \\ \mu g \equiv \varepsilon x.\ x = g\,x \land (\forall y\,z.\ y = g\,y \land z = g\,z \longrightarrow y = z) \end{array}$$

The expression μg represents the unique fixed point of g, when one exists. If g does not have a *unique* fixed point, then μg denotes an arbitrary value.

3.2 Properties of Unique Fixed Points

As an aside, several nice properties hold when one can establish that a system of recursive equations has a unique solution. For example, unique fixed points can sometimes "absorb" functions applied to other fixed points.

Lemma 1 *Given functions $F : \alpha \to \alpha$, $G : \beta \to \beta$, $f : \alpha \to \beta$, and value $x : \alpha$, such that x is a (not necessarily unique) fixed point of F, G has a unique fixed point, and $f \circ F = G \circ f$, then $f\,x = \mu G$.*

Unique fixed points can also be "rotated", in the following sense:

Lemma 2 *If the composition of two functions $g : \alpha \to \beta$ and $h : \beta \to \alpha$ has a unique fixed point $\mu(g \circ h)$, then $h \circ g$ also has a unique fixed point, and $\mu(g \circ h) = g(\mu(h \circ g))$.*

Although we will not use Lemma 1 or Lemma 2 in the remainder of the paper, lemmas such as these are useful for manipulating systems of recursive equations as objects in their own right.

4 Converging Equivalence Relations and Contracting Functions

While unique fixed points are a useful definition mechanism, it can be difficult to show that they exist for a given function. A direct proof usually involves constructing an explicit fixed point witness using other definition techniques, such as corecursion or well-founded recursion. Little effort seems to be saved.

We propose an alternative proof technique, based on concepts from domain theory[4, 15] and topology[1, 11] where one builds a collection of ever-closer approximations to the desired fixed point, and shows that the limit of these approximations exists, is a fixed point of the function under consideration, and is unique. The approximation process can be parameterized to some extent, and reused across multiple definitions that are \similar" enough. Furthermore these parameterized approximations can be composed hierarchically, yielding more powerful approximation techniques.

4.1 Converging Equivalence Relations

To make the notion of approximation precise, we need a way of stating how \close" two potential approximations are to each other. One approach would be to define a suitable metric space[1] and use the corresponding distance function, which returns either a rational or real number, given any two elements in the domain of the metric space. However, proving that a series of approximations converges to a limit point often requires one to reason about exponentiation and division over a theory of rationals or reals. An alternative way to measure \closeness", which we call a *converging equivalence relation* (CER), instead only involves reasoning about well-founded sets, such as the set of natural numbers, or the set of finite lists. In many cases we can prove a unique fixed point exists by performing a simple induction over the natural numbers, something which all of the current HOL theorem provers support well.

A converging equivalence relation consists of:

- A type , called the *resolution space*
- A type , called the *target space*
- A well-founded, transitive relation (<) over type , called a *resolution ordering*
- A three-argument predicate ( ) of type (*!* *!* *!* *bool*), called an *indexed equivalence relation*. Given an element *i* of type , and two elements *x* and *y* of type , we denote the application of ( ) to *i*, *x* and *y* as ($x \overset{i}{\;} y$), and if this value is true, then we say that *x* and *y* are *equivalent at resolution i*.

The resolution ordering (<) and indexed equivalence relation ( ) must satisfy the properties in Fig. 1, for arbitrary $i; i^0$: ; $x; y; z$: ; and f : ! . Axioms (1), (2), and (3) state that ( ) must be an equivalence relation at each resolution i. Axiom (4) states that if a resolution i has no lower resolutions, then ( ) treats all target elements as equivalent at that resolution. Such resolutions are called *minimal.* There is always at least one minimal resolution (and perhaps more than one), since (<) is well-founded. Axiom (5) states that if two elements are equivalent at a particular resolution, then they are equivalent at all lower resolutions. Thus higher resolutions impose finer-grained, but compatible, partitions of the target space than lower resolutions do. Although no particular resolution may distinguish all elements, (6) states that if two elements are equivalent at all resolutions, then they are in fact equal.

$$x \ ^{i}\ x \tag{1}$$

$$x \ ^{i}\ y - !\ y \ ^{i}\ x \tag{2}$$

$$x \ ^{i}\ y \,\hat{}\, y \ ^{i}\ z - !\ x \ ^{i}\ z \tag{3}$$

$$(8j : : (j < i)) - !\ x \ ^{i}\ y \tag{4}$$

$$x \ ^{i'}\ y \,\hat{}\, i < i' - !\ x \ ^{i}\ y \tag{5}$$

$$(8j : x \ ^{j}\ y) - !\ x = y \tag{6}$$

$$(8j; j' : j < j' < i - !\ (f\,j) \ ^{j}\ (f\,j')) - !\ (9z : 8j < i : z \ ^{j}\ (f\,j)) \tag{7}$$

$$(8j; j' : j < j' - !\ (f\,j) \ ^{j}\ (f\,j')) - !\ (9z : 8j : z \ ^{j}\ (f\,j)) \tag{8}$$

Fig. 1. The CER axioms. Each of these axioms must hold for arbitrary i, x, y, and f.

Axioms (7) and (8) deal with \limits" of approximations. First some terminology: a function f : ! from the space of resolutions to the target space of elements is called an *approximation map*. An approximation map f is *convergent up to resolution i* if for all resolutions j and j^0 such that $j < j^0 < i$, then $(f\,j)$ is equivalent at resolution j to $(f\,j^0)$. Note that it is possible for $(f\,i)$ itself not to be equivalent to any of the lower-resolution $(f\,j)$'s. An approximation map f is *globally convergent* if for all resolutions j and j^0 such that $j < j^0$, then $(f\,j) \ ^{j}\ (f\,j^0)$.

Axiom (7) states that if f is locally convergent up to resolution i, then there exists a limit-like element z that is equivalent at each resolution $j < i$ to the corresponding $(f\,j)$ approximation (there may be multiple such elements). Axiom (8) states that if f is globally convergent, then there exists a limit element z that is equivalent to each approximation $(f\,j)$ at resolution j.

4.2 Examples of Converging Equivalence Relations

Discrete CER The simplest useful CER has as a resolution space a two-element type containing the values $?$ and $>$, with $(? < >)$, and a target space with ( ) defined such that $(x \overset{?}{\;} y)$ $True$, and $(x \overset{>}{\;} y)$ $(x = y)$. Axioms (1) through (6) are easy to verify. Axiom (7) holds for any element. The limit element satisfying (8) is $f\ >$.

Lazy List CER We can construct a converging equivalence equation for comparing coinductive lists by comparing the first i elements of two lazy lists l_1 and l_2 at a given resolution i. To perform the comparison, we make use of the *ltake* function, with type $nat\ \ !\ \ \ llist\ \ !\ \ \ list$. The expression $(ltake\ n\ xs)$ returns a finite list consisting of the first n elements of xs. If xs has fewer than n elements, then *ltake* returns the whole of xs. The *ltake* function can be defined by well-founded recursion on its numeric argument with the following recursive equations:

$$\begin{array}{llll} ltake & 0 & xs & = & [] \\ ltake & (n+1) & [] & = & [] \\ ltake & (n+1) & (x \# xs) & = & x \# (ltake\ n\ xs) \end{array}$$

We then define the lazy list CER with the natural numbers as the resolution space, ($llist$) as the target space, the usual ordering on the natural numbers for $(<)$, and ( ) defined as follows:

$$xs \overset{i}{\;} ys \qquad (ltake\ i\ xs = ltake\ i\ ys):$$

Axioms (1) through (3) hold trivially. The only minimal resolution in this CER is 0, and since $(ltake\ 0\ xs) = []$, then (4) holds. If two lazy lists are equal up to the first i positions, then they are equal up to any $i^0 < i$ position, so (5) holds. Axiom (6) reduces to the Take Lemma[10], which can be proved by coinduction.

Axioms (7) and (8) require us to construct appropriate limit elements, given an approximation map. Both limit elements can be constructed by a single function, which we call *llist_diag*. For a given approximation map f, the limit elements may be of infinite length, so we define *llist_diag* by corecursion, using *llist_corec*:

$$\begin{array}{l} llist\_diag\ f \qquad llist\_corec\ 0\ (nthElem\ f) \\ where \\ \qquad nthElem\ f\ n \qquad \begin{cases} \mathrm{Inl}\,(); & \text{if } ldrop\ n\ (f\ (n+1)) = [] \\ \mathrm{Inr}\,(x; n+1); & \text{if } ldrop\ n\ (f\ (n+1)) = (x \# xs) \end{cases} \end{array}$$

The helper function *nthElem* uses the *ldrop* function on lazy lists. The *ldrop* function has type $nat\ \ !\ \ (\ \ llist)\ \ !\ \ (\ \ llist)$, and $(ldrop\ i\ xs)$ removes the first i elements from xs, returning the remainder. Like *ltake*, it is defined by well-founded recursion on its numeric argument:

$$\begin{array}{lllll} ldrop & 0 & xs & = & xs \\ ldrop & (n+1) & [] & = & [] \\ ldrop & (n+1) & (x \# xs) & = & ldrop\ n\ xs \end{array}$$

The overall action of *llist_diag* is to construct a so-called *diagonal list* from the approximation map f, where the n^{th} element of the result list is drawn from the n^{th} element of approximation $f\,(n+1)$, if the n^{th} element exists. If the n^{th} element does not exist (i.e., the length of $f\,(n+1)$ is less than n), then the result list is terminated at that point.

It turns out that for any CER whose $(<)$ relation is the less-than ordering on the natural numbers, the following property implies both (7) and (8):

$$\forall f.(\forall i.(f\ i) \approx_i (f\ (i+1))) \longrightarrow (\exists x.\forall i.x \approx_i (f\ i)).$$

With some work, one can show that this property holds for the lazy list CER by supplying *llist_diag f* as the existential witness element for x.

4.3 Contracting Functions

In the theory of metric spaces, a *contracting function* is a function F such that for any two points x and y, $F\,x$ is closer to $F\,y$ than x is to y, given a suitable distance function. Banach's theorem states that all contracting functions over suitable metric spaces have unique fixed points. We can define an analogous notion over a CER:

Definition 1 *A function F is* contracting *over a CER given by $(<)$ and $(\approx)$ if for all resolutions i and target elements x and y,*

$$(\forall i' < i.x \approx_{i'} y) \longrightarrow (F\ x) \approx_i (F\ y).$$

Intuitively a function is contracting if, given two elements x and y that are close enough together at all lower resolutions $i' < i$ to satisfy the CER, but are potentially too far away at resolution i, then F maps them to two elements that are now close enough at resolution i.

For example, the function $\mathit{consZero}\ \mathit{xs} \equiv (0\,\#\,\mathit{xs})$ is contracting over the lazy list CER, since given any i and two lazy lists xs and ys,

$$(\forall i' < i.\mathit{ltake}\ i'\ \mathit{xs} = \mathit{ltake}\ i'\ \mathit{ys}) \longrightarrow \mathit{ltake}\ i\ (\mathit{consZero}\ \mathit{xs}) = \mathit{ltake}\ i\ (\mathit{consZero}\ \mathit{ys}).$$

The main result of this paper is as follows:

Theorem *A contracting function F over a CER has a unique fixed point.*

The proof is discussed in Sect. 7. For now, we would like to apply this theorem to define some simple recursive functions over lazy lists.

4.4 Recursive Definitions over Coinductive Lists

To begin with, we can simplify the definition of a contracting function F over a CER when the $(<)$ relation of that CER is the less-than relation over the natural numbers. In this case, Definition 1 reduces to

$$\forall i\,x\,y.x \approx_i y \longrightarrow (F\ x) \approx_{i+1} (F\ y). \qquad (9)$$

Specializing this formula for the lazy list CER, we have that F is contracting on lazy lists if

$$\forall i\, x\, y.\ \mathit{ltake}\ i\ x = \mathit{ltake}\ i\ y \longrightarrow \mathit{ltake}\ (i+1)\ (F\ x) = \mathit{ltake}\ (i+1)\ (F\ y). \qquad (10)$$

Defining Iterates Let us establish that the following recursive equation, defined over x and f, has a unique solution, and is thus a definition:

$$\mathit{iterates} = (x \mathbin{\#} (\mathit{lmap}\ f\ \mathit{iterates})) \qquad (11)$$

This equation builds the infinite list $[x, f\ x, f\ (f\ x), \ldots]$. We first define the non-recursive functional F that characterizes this equation:

$$F\ \mathit{iterates}' \equiv (x \mathbin{\#} (\mathit{lmap}\ f\ \mathit{iterates}')).$$

and then show that it is a contracting function. To do this we rely on (10), and assume we have two arbitrary lazy lists xs and ys such that $\mathit{ltake}\ i\ xs = \mathit{ltake}\ i\ ys$. We now need to show that $\mathit{ltake}\ (i+1)\ (F\ xs) = \mathit{ltake}\ (i+1)\ (F\ ys)$. Using a process of equational simplification we are able to reduce the goal to the assumption, as follows:

$$\begin{aligned}
& \mathit{ltake}\ (i+1)\ (F\ xs) = \mathit{ltake}\ (i+1)\ (F\ ys)\\
\Longleftrightarrow\ & \mathit{ltake}\ (i+1)\ (x \mathbin{\#} (\mathit{lmap}\ f\ xs)) = \mathit{ltake}\ (i+1)\ (x \mathbin{\#} (\mathit{lmap}\ f\ ys))\\
\Longleftrightarrow\ & \mathit{ltake}\ i\ (\mathit{lmap}\ f\ xs) = \mathit{ltake}\ i\ (\mathit{lmap}\ f\ ys)\\
\Longleftarrow\ & \mathit{ltake}\ i\ xs = \mathit{ltake}\ i\ ys
\end{aligned}$$

The simplification relies on the following facts, each proved by induction on i:

$$\begin{gathered}
(\mathit{ltake}\ (i+1)\ (z \mathbin{\#} xs) = \mathit{ltake}\ (i+1)\ (z \mathbin{\#} ys)) \Longleftrightarrow (\mathit{ltake}\ i\ xs) = \mathit{ltake}\ i\ ys)\\
(\mathit{ltake}\ i\ (\mathit{lmap}\ f\ xs) = \mathit{ltake}\ i\ (\mathit{lmap}\ f\ ys)) \Longleftarrow (\mathit{ltake}\ i\ xs = \mathit{ltake}\ i\ ys)
\end{gathered}$$

These facts illustrate a nice property of this proof: We did not have to expand the definitions of (#) or *lmap* during the simplification process, relying instead on an abstract characterization of their behavior with respect to *ltake*. This turns out to be the case for many functions, even recursive ones defined by contracting functions. In general we can often incrementally define recursive functions and prove properties about how they behave with respect to ( ), without having to expand the definitions of functions making up the body of the recursive definition.

5 Composing Converging Equivalence Relations

The lazy list CER allows us to give recursive definitions of individual lazy lists, but we are often more interested in recursively defining functions that transform lazy lists. Fortunately, there are several *CER combinators* that allow us to build CERs over complex types, if we have CERs that operate on the corresponding atomic types.

Local and Global Limits When constructing a new CER C' out of an existing CER C, we usually have to show (7) and (8) hold for C' by invoking (7) and (8) for C, to create the necessary limit witness elements. To make this process explicit, we use Hilbert's description operator (ε) to create functions that return these witness elements[3], given an appropriate approximation mapping f:

$$local\_limit :: (\quad \to \quad) \to \quad \to$$
$$local\_limit\ f\ i \quad (\varepsilon z.\ \forall j < i.\ z\ {}^{j}\ (f\ j)) \qquad (12)$$

$$global\_limit :: (\quad \to \quad) \to$$
$$global\_limit\ f \quad (\varepsilon z.\ \forall j.\ z\ {}^{j}\ (f\ j)) \qquad (13)$$

We can use (7) and (8) to prove the basic properties we want *local_limit* and *global_limit* to have for any CER given by ($<$) and ( ):

$$(\forall j, j'.\ j < j' < i \longrightarrow (f\ j)\ {}^{j}\ (f\ j')) \longrightarrow (\forall j < i.\ (local\_limit\ f\ i)\ {}^{j}\ (f\ j))$$
$$(\forall j, j'.\ j < j' \longrightarrow (f\ j)\ {}^{j}\ (f\ j')) \longrightarrow (\forall j.\ (global\_limit\ f)\ {}^{j}\ (f\ j))$$

Function-Space CER The functions *local_limit* and *global_limit* allow us to concisely specify the limit elements of CER combinators. For example, given a CER C from resolution space to target space given by ($<$) and ( ), we can construct a new *function-space over C* CER with the same resolution ordering ($<$), and a new indexed equivalence relation ($'$) with type $\to (\quad \to \quad) \to (\quad \to \quad) \to bool$, defined as

$$g\ {}^{i}{}'\ h \quad \forall x.\ (g\ x)\ {}^{i}\ (h\ x).$$

The limit elements satisfying (7) and (8) can be given as

$$local\_limit\_fun\ f\ i \quad (\ x.\ local\_limit\ (\ i.\ f\ i\ x)\ i)$$
$$global\_limit\_fun\ f \quad (\ x.\ global\_limit\ (\ i.\ f\ i\ x))$$

Given these limit-producing functions, is relatively easy to show that the function-space over C CER satisfies the CER axioms.

5.1 Defining Recursive Functions with the Function-Space CER

Defining lmap We can apply the function-space CER to define *lmap* recursively. The recursion equations for *lmap* are:

$$lmap\ f\ [] = []$$
$$lmap\ f\ (x\#xs) = (f\ x)\ \#\ (lmap\ f\ xs)$$

[3] This is merely a convenience. The CER properties can be shown with a little more work in Isabelle using (7) and (8) directly.

We translate the equations into a non-recursive form (parameterized over f)

$$\begin{aligned} F\ lmap' \equiv (\lambda xs.\ &\text{case}\ xs\ \text{of} \\ &[] \Rightarrow [] \\ &\mid (y \# ys) \Rightarrow (f\ y) \# (lmap'\ ys)). \end{aligned}$$

We then need to show that fix F is the unique fixed point of F by proving that F is a contracting function on the function-space over lazy lists CER. By (9) we must show for arbitrary resolution i and functions g and h, that $(g \overset{i}{\approx}' h \longrightarrow (F\ g) \overset{(i+1)}{\approx}' (F\ h))$. Expanding definitions, we obtain

$$\begin{aligned} & g \overset{i}{\approx}' h \longrightarrow (F\ g) \overset{(i+1)}{\approx}' (F\ h) \\ \Leftrightarrow\ & (\forall xs.\ g\ xs \overset{i}{\approx} h\ xs) \longrightarrow (\forall xs.\ (F\ g\ xs) \overset{(i+1)}{\approx} (F\ h\ xs)) \\ \Leftrightarrow\ & (\forall xs.\ ltake\ i\ (g\ xs) = ltake\ i\ (h\ xs)) \longrightarrow \\ & \quad (\forall xs.\ ltake\ (i+1)\ (F\ g\ xs) = ltake\ (i+1)\ (F\ h\ xs)). \end{aligned}$$

So, to prove F is contracting we take an arbitrary resolution i and two arbitrarily chosen functions g and h such that $(\forall xs.\ ltake\ i\ (g\ xs) = ltake\ i\ (h\ xs))$, and show for an arbitrary xs that $ltake\ (i+1)\ (F\ g\ xs) = ltake\ (i+1)\ (F\ h\ xs)$. There are two cases to consider:

case $xs = []$:

$$\begin{aligned} & ltake\ (i+1)\ (F\ g\ []) = ltake\ (i+1)\ (F\ h\ []) \\ \Leftrightarrow\ & ltake\ (i+1)\ [] = ltake\ (i+1)\ [] \\ \Leftrightarrow\ & \text{True.} \end{aligned}$$

case $xs = (y \# ys)$:

$$\begin{aligned} & ltake\ (i+1)\ (F\ g\ (y \# ys)) = ltake\ (i+1)\ (F\ h\ (y \# ys)) \\ \Leftrightarrow\ & ltake\ (i+1)\ ((f\ y) \# (g\ ys)) = ltake\ (i+1)\ ((f\ y) \# (h\ ys)) \\ \Leftrightarrow\ & ltake\ i\ (g\ ys) = ltake\ i\ (h\ ys) \\ \Leftrightarrow\ & \text{True } \{\text{by assumption}\}. \end{aligned}$$

Given the definition of F and basic lemmas about *ltake*, Isabelle's high-level simplification tactics allow the above proof to be carried out in two steps. The proof completes in about a second on a 266MHz Pentium II.

Defining lappend We can apply the function-space CER combinator repeatedly, to prove that multi-argument curried functions have unique fixed points. As a concrete example, the curried function *lappend* has type $\alpha\ llist \to \alpha\ llist \to \alpha\ llist$. It takes two lazy list arguments xs and ys and returns a new list consisting of the elements of xs followed by the elements of ys. The recursive equations for *lappend* are

$$\begin{aligned} lappend\ []\ ys &= ys \\ lappend\ (x \# xs)\ ys &= (x \# lappend\ xs\ ys) \end{aligned}$$

To prove that these equations have a unique solution, we apply the function-space CER combinator to the lazy list CER to obtain a new CER C'. We then apply the function-space CER combinator again to C', obtaining a new CER C'' with the usual less-than relation on *nat* for $(<)$ and the following indexed equivalence relation $(\approx'')$:

$$g \approx''^{i} h \equiv (\forall xs\ ys.\ ltake\ i\ (g\ xs\ ys) = ltake\ i\ (h\ xs\ ys)).$$

Next, we convert the recursive equations for *lappend* into a non-recursive function F:

$$\begin{aligned} F\ lappend' \equiv (\lambda\ xs\ ys.\ & \text{case}\ xs\ \text{of} \\ & \quad [] \Rightarrow ys \\ & \mid (x \# xs') \Rightarrow (x \# (lappend'\ xs'\ ys))). \end{aligned}$$

By (9) we must show for arbitrary resolution i and functions g and h, that

$$\begin{aligned} &(\forall xs\ ys.\ ltake\ i\ (g\ xs\ ys) = ltake\ i\ (h\ xs\ ys)) \longrightarrow \\ &\quad (\forall xs\ ys.\ ltake\ (i+1)\ (F\ g\ xs\ ys) = ltake\ (i+1)\ (F\ h\ xs\ ys)). \end{aligned}$$

So we take arbitrary i, xs, and ys, and prove

$$ltake\ (i+1)\ (F\ g\ xs\ ys) = ltake\ (i+1)\ (F\ h\ xs\ ys)$$

assuming we have $(\forall xs\ ys.\ ltake\ i\ (g\ xs\ ys) = ltake\ i\ (h\ xs\ ys))$. There are two cases to consider, depending on whether xs is empty or not:

case $xs = []$:

$$\begin{aligned} & ltake\ (i+1)\ (F\ g\ []\ ys) = ltake\ (i+1)\ (F\ h\ []\ ys) \\ \Leftrightarrow\ & ltake\ (i+1)\ ys = ltake\ (i+1)\ ys \\ \Leftrightarrow\ & \text{True}. \end{aligned}$$

case $xs = (x \# xs')$:

$$\begin{aligned} & ltake\ (i+1)\ (F\ g\ (x \# xs')\ ys) = ltake\ (i+1)\ (F\ h\ (x \# xs')\ ys) \\ \Leftrightarrow\ & ltake\ (i+1)\ (x \# (g\ xs'\ ys)) = ltake\ (i+1)\ (x \# (h\ xs'\ ys)) \\ \Leftrightarrow\ & ltake\ i\ (g\ xs'\ ys) = ltake\ i\ (h\ xs'\ ys) \\ \Leftrightarrow\ & \text{True}\ \{\text{by assumption}\}. \end{aligned}$$

Thus we can conclude that *lappend* has a unique fixed point definition. We were able to carry out this proof in Isabelle in three steps, again taking about a second of CPU time.

5.2 Other CER Combinators

CER combinators can also be defined over product and sum types. The lazy list CER can be generalized to work over any coinductive type that has a notion of depth, such as coinductive trees. A more powerful function-space CER is discussed in Sect. 6.

5.3 Demonstrating Equality between Coinductive Elements

Converging equivalence relations can also be useful in showing that two elements of a target space are equal. Axiom (6) (restated below) says that to show two target elements x and y are equal, one simply needs to show they are equivalent at all resolutions j

$$(\forall j.\, x \simeq^{j} y) \longrightarrow x = y.$$

We can often demonstrate that x and y are equivalent at all resolutions by well-founded induction, since $(<)$ is a well-founded relation. For example, given two arbitrary lazy lists ys and zs, we can prove the following lemma about *lappend* by (simple) induction on i, followed by a case split on xs:

Lemma 3

$$\forall xs.\, ltake\ i\ (lappend\ (lappend\ xs\ ys)\ zs) = ltake\ i\ (lappend\ xs\ (lappend\ ys\ zs)).$$

The proof takes four steps in Isabelle. Given (6) instantiated to the lazy list CER, we can then easily show in one Isabelle step that $lappend\ (lappend\ xs\ ys)\ zs = lappend\ xs\ (lappend\ ys\ zs)$.

6 Defining Functions with Unbounded Look-Ahead

The functions we have defined so far examine their arguments by performing at most one pattern match on a lazy list before producing an element of a result list. However, there is a class of functions that can examine a potentially infinite amount of their argument lists before deciding the next element to output. An example is the *lazy filter* function of type $(\alpha \Rightarrow bool) \Rightarrow \alpha\ llist \Rightarrow \alpha\ llist$, which takes a predicate P and a lazy list xs, and returns a lazy list of the same type consisting only of those elements of xs satisfying P. A candidate set of recursion equations for this function might be

$$\begin{array}{lll} lfilter\ P\ [] & = [] & \\ lfilter\ P\ (x \# xs) & = lfilter\ P\ xs; & \text{if } \neg(P\ x) \\ lfilter\ P\ (x \# xs) & = x \# (lfilter\ P\ xs); & \text{if } P\ x \end{array}$$

Sadly, this intuitively appealing set of equations does not completely define *lfilter*. If *lfilter* is given an infinite list xs, none of whose elements satisfy P, then the above equations do not specify what the result list should be. The *lfilter* function is free to return any value at all in this case. In other words, the equations do not have a unique solution.

Happily we can remedy the situation as follows: We define by induction over *nat* a predicate *firstPelemAt* of type $(\alpha \Rightarrow bool) \Rightarrow \alpha\ llist \Rightarrow nat \Rightarrow bool$. The expression $(firstPelemAt\ P\ xs\ i)$ is true if xs has at least $(i + 1)$ elements and i

is the position of the first element of *xs* satisfying P. We can then define the predicate *never* of type ($!$ *bool*) $!$ *llist* $!$ *bool* as

$$never\ P\ xs \quad 8\, i :\ : (\ firstPelemAt\ P\ xs\ i)$$

which is true when there are no elements in *xs* satisfying P. If we modify the initial recursive equations as follows:

$$\begin{array}{lll} filter\ P\ xs & = [], & \text{if } never\ P\ xs \\ filter\ P\ (x\#xs) & = filter\ P\ xs; & \text{if } :(never\ P\ xs)\ \wedge\ :(P\ x) \\ filter\ P\ (x\#xs) & = x\ \#\ (filter\ P\ xs); & \text{if } :(never\ P\ xs)\ \wedge\ P\ x \end{array}$$

then the set of equations does indeed have a unique solution. This function is not computable, since the predicate *never* can scan an infinite number of elements, but it is nevertheless mathematically valid in HOL. The CERs described above are not powerful enough to prove this, but we can define a *well-founded function-space* CER combinator that is. Given a CER C with ($<$) of type $!$ $!$ *bool* and ( ) with type $!$ $!$ $!$ *bool*, and another well-founded transitive relation ( ) of type $!$ $!$ *bool*, we define our new CER C^0 with ($<^0$) and (^{0}) as follows:

$$\begin{array}{l} (<^0) :: (\quad)\ !\ (\quad)\ !\ bool \\ (\ ^0) :: (\quad)\ !\ (\ !\ )\ !\ (\ !\ )\ !\ bool \end{array}$$

$$\begin{array}{ll} (a^0; t^0) <^0 (a; t) & a^0 < a\ \_\ (a^0 = a\ \wedge\ t^0\ \ t) \\ g \overset{(a;t)}{\ ^0} h & 8\, a^0\, t^0 : (a^0; t^0)\ ^0\ (a; t) \longrightarrow\!!\ (g\, t^0) \overset{a'}{\ } (h\, t^0) \end{array}$$

It is a fair amount of work to show that C^0 is in fact a CER, and space constraints force us to elide the details.

Intuitively, however, C^0 allows us to generalize well-founded recursion in the following way: A well-founded recursive function is forced to have its argument decrease in size on every recursive call. With C^0, the function being defined is allowed a choice; it can either decrease the size of its argument when making a recursive call, or not decrease its argument size but then make sure the element it is returning is \larger" than the element returned from its recursive call.

In the case of functions returning lazy lists, a \larger" lazy list is one that looks just like the lazy list returned by the recursive call, but with at least one extra element added to the front.

For us to use C^0 on *filter*, we need to specify a suitable well-founded transitive relation ( ). The relation we choose is one that holds when the first element satisfying P occurs sooner on the left-hand argument than on the right-hand argument:

$$\begin{array}{l} xs\ \ ys \quad firstPelem\ P\ xs\ <\ firstPelem\ P\ ys \\ \text{where} \\ \qquad firstPelem\ P\ xs\ = 0; \qquad \text{if } never\ P\ xs \\ \qquad\qquad = 1 + ("i :\ firstPelemAt\ P\ xs\ i), \quad \text{otherwise} \end{array}$$

We arbitrarily decide that a list containing no P-elements is -smaller than any list with at least one P-element.

When analyzing the revised recursive equations for *lfilter*, if *xs* has no P-elements then we return immediately, otherwise xs has to have at least one P-element. If that element is not at the head of the list, then the tail of the list is -smaller than *xs*. If the first P-element is at the head of *xs*, then the tail of the list is not -smaller than *xs*, but the output list has one more element than the list returned by the recursive call. Thus we informally conclude that *lfilter* is uniquely defined.

We have also proved this fact formally in Isabelle. After inductively proving various simple lemmas about *firstPelemAt*, *never*, and *firstPelem*, we were able to prove that *lfilter* is uniquely defined in five steps. We first translated the recursive equations above into a contracting function F. We used C^0 prove that F is contracting, first by expanding the definition of F and simplifying, and then by performing a case analysis (no induction required!) on whether the *nat* component of the current resolution was equal to zero. It took Isabelle two seconds to perform the proof.

Although we had to prove lemmas about *firstPelemAt*, *never*, and *firstPelem*, the proofs are not hard and it turns out we can reuse these results when defining other functions that perform unbounded search on lazy lists. For example, the *lflatten* function takes a lazy list of lazy lists, and flattens all of the elements into a single lazy list. The *lflatten* function can also be uniquely defined using *never*:

$$\begin{array}{lll} \mathit{lflatten}\ \mathit{xss} & = [], & \text{if } \mathit{never}\ (\ \mathit{xs} : \mathit{xs} \neq [])\ \mathit{xss} \\ \mathit{lflatten}\ (\mathit{xs} \# \mathit{xss}) & = \mathit{lappend}\ \mathit{xs}\ (\mathit{lflatten}\ \mathit{xss}); & \text{otherwise} \end{array}$$

The proof proceeds in Isabelle exactly as it does for *lfilter* except that we perform one additional case analysis on whether $xs = []$. The proof takes three seconds to complete.

7 Proof of the Main Result

Although the proof of the main theorem is too lengthy to describe here, we will provide a rough outline. Given a CER with resolution space , target space , well-founded relation ($<$), indexed equivalence relation ( ), and an arbitrary contracting function F of type ! , the technique will be to construct an approximation map *apx* F that converges globally to the desired fixed point. We then prove that this fixed point is unique by showing that any two fixed points of F are equal.

The function *apx* of type (!) ! ! that builds an approximation map from a contracting function is defined by well-founded recursion on ($<$) as follows:

$$\mathit{apx}\ F\ i \quad F\ (\mathit{local\_limit}\ (\mathit{cut}\ (\mathit{apx}\ F)\ i)\ i)$$

where

$$\mathit{cut}\ f\ i\ x \quad \text{if } x < i \text{ then } f\ x \text{ else } \mathit{arbitrary}.$$

At each resolution i, the function apx uses *local_limit* to obtain the best possible approximation of xF, given the approximations it has already computed at all lower resolutions[4]. The result of calling *local_limit* may still not be close enough at resolution i, so *apx* maps the local limit through F, which will bring the result close enough. The helper function *cut* is used to ensure that the recursive call to *apx* F is only made at lower resolutions than i, ensuring well-foundedness. If *local_limit* attempts to invoke *cut* (*apx* F) i at any other resolution, then *cut* returns an arbitrary element instead.

Once we have proved by well-founded induction that apx is well defined, the next step is to establish that $apx\,F$ is convergent up to each resolution i. To do this we prove several lemmas, such as: if an approximation mapping f converges up to a local limit element z at resolution i, and also converges up to a local limit element z^0 at the same resolution, then z and z^0 are equivalent at all resolutions $i^0 < i$. With this, and the fact that F is contracting, we can show that if $x \;^{i}\; y$, then $F\,x \;^{i}\; F\,y$. We then eventually show for all resolutions i that if *apx* F converges up to local limit element *apx* $F\ i$ at resolution i, then *apx* $F\ i \;^{i}$ F (*apx* $F\ i$). This lemma is the key to showing by well-founded induction over i that $apx\,F$ does in fact converge up to *apx* $F\ i$ at resolution i, and is also used to show that *global_limit* (*apx* F) ^{i} F (*global_limit* (*apx* F)) at each resolution i, and are thus equal by (6). This result establishes that a fixed point exists for F. We then show that any two fixed points x and y of F are equivalent at all resolutions by well-founded induction, and thus are equal, again by (6).

8 Conclusion

Related Work The support for and application of well-founded induction and general coinduction has seen wide acceptance in the HOL theorem proving community. The well-founded definition package TFL used in HOL98 and Isabelle was written by Slind[13]. It can handle nested pattern matching in rule definitions, nested recursion in function bodies, and generates custom induction rules for each definition[14]. The PVS theorem prover[12] also uses well-founded induction as a basic definitional principle. A general theory of inductive and coinductive sets in Isabelle was developed by Paulson[10], based on least and greatest fixed points of monotone set-transforming functions, as well as a package for defining new inductive and coinductive sets by user-given introduction rules. The package avoids syntactic restrictions in the introduction rules by reasoning about each rule's underlying set-transformer semantics.

A coinductive theory of streams (infinite-only lists) was developed by Miner[7] in the PVS theorem prover. Miner used this theory to model synchronous hardware circuits as corecursively-defined stream transformers. Using coinduction, he was able to optimize the implementation of a fault-tolerant clock synchronization circuit and a floating-point division circuit.

[4] Here the definition of *local_limit* using Hilbert's choice operator seems essential.

A well-known alternative to coinductive types is the mathematical framework of *pointed complete partial orders* and *continuous functions*, also known as *domain theory*[4, 15]. This theory is supported by the HOLCF[8] object-logic in Isabelle, and also allows one to define infinite data structures such as lazy lists and trees. A wide variety of functions over these structures can then be recursively defined. The primary disadvantage of this approach is that one must add \extra" bottom-elements to the structures being defined. These extra elements are used to indicate that a function is non-terminating on its arguments. For example, the lazy filter function *lfilter* can be defined recursively in HOLCF, but the expression *lfilter* P *xs* returns $?$ instead of [] when *xs* is an infinite list containing no elements satisfying P. Also, only so-called *admissible* predicates can be reasoned about inductively in domain theory, and it can be quite challenging to prove that a desired predicate is admissible. A comparison of the HOLCF approach to several other encodings of lazy lists is presented by Devillers et al[2].

The theory of topology[1, 11] provides another well-established definition mechanism. The notions of Cauchy sequences, complete metric spaces, and contractions inspired much of this work. We have not worked out the exact relationship between converging equivalence relations and Cauchy metric spaces; although one can construct a distance function for every *nat*-indexed CER, it is not clear that distance functions can always be constructed for more complex resolution spaces. Also, the conditions under which a function F is contracting in a CER seem to be less restrictive than the corresponding conditions in a metric space. More importantly from a verification perspective, well-founded induction seems easier to apply in current theorem provers than does the continuous mathematics required for metric spaces.

Current and Future Work We are currently using CERs to specify and reason about processor microarchitectures as recursively defined stream transformers. This work is part of the Hawk project[6], which is developing a domain-specific functional language for specifying, simulating, and reasoning about such microarchitectures at a high level of abstraction. We have been able to use CERs and the unique fixed point lemmas in Sect. 3.2 to develop a domain-specific *microarchitecture algebra*[5] in Isabelle, which we use to verify Hawk specifications.

Although we have defined CERs over streams and lazy lists, many structures in language semantics and process algebras can be seen as coinductive trees. It would be interesting to define some of these structures recursively and reason about them inductively, as we did for *lappend* in Sect. 5.3.

9 Acknowledgements

We wish to thank Byron Cook, Sava Krstic, and John Launchbury for their valuable contributions to this research. We are also grateful for the anonymous referees' helpful suggestions. The author is supported by a graduate research fellowship with the National Science Foundation, and grants from the Air Force Material Command (F19628-93-C-0069) and Intel Strategic CAD Labs.

References

[1] Buskes, G., and van Rooij, A. *Topological Spaces: from distance to neighborhood.* UTM Series. Springer, New York, 1997.

[2] Devillers, M., Griffioen, D., and Müller, O. Possibly infinite sequences in theorem provers: A comparative study. In *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97* (Murray Hill, NJ, Aug. 1997), vol. 1275 of *LNCS*, Springer-Verlag, pp. 89{104.

[3] Gordon, M. J. C., and Melham, T. F. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[4] Gunter, C. A. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Science. The MIT Press, 1992.

[5] Matthews, J., and Launchbury, J. Elementary microarchitecture algebra. To appear in CAV99, International Conference on Computer Aided Verification, July 1999.

[6] Matthews, J., Launchbury, J., and Cook, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 90{101.

[7] Miner, P. *Hardware Verification Using Coinductive Assertions.* PhD thesis, Indiana University, 1998.

[8] Müller, O., Nipkow, T., v. Oheimb, D., and Slotosch, O. HOLCF = HOL + LCF. To appear in Journal of Functional Programming, 1999.

[9] Paulson, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

[10] Paulson, L. C. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation 7*, 2 (Apr. 1997), 175{204.

[11] Rudin, W. *Principles of Mathematical Analysis*, 3 ed. McGraw-Hill, 1976.

[12] Rushby, J., and Stringer-Calvert, D. W. J. A less elementary tutorial for the PVS specification and verification system. Tech. Rep. SRI-CSL-95-10, SRI International, Menlo Park, CA, June 1995. Revised, July 1996.

[13] Slind, K. Function definition in higher order logic. In *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL* (Turku, Finland, Aug. 1996), J. Von Wright, J. Grundy, and J. Harrison, Eds., vol. 1125 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 381{398.

[14] Slind, K. Derivation and use of induction schemes in higher-order logic. *Lecture Notes in Computer Science 1275* (1997), 275{290.

[15] Tennent, R. D. *Semantics of Programming Languages.* Prentice Hall, New York, 1991.

Hardware Verification Using Co-induction in COQ

Solange Coupet{Grimal and Line Jakubiec

Laboratoire d'Informatique de Marseille { URA CNRS 1787
39, rue F. Joliot{Curie 13453 Marseille France
fSolange.Coupet,Line.Jakubiecg@lim.univ-mrs.fr

Abstract. This paper presents a toolbox implemented in Coq and dedicated to the specification and verification of synchronous sequential devices. The use of Coq co-inductive types underpins our methodology and leads to elegant and uniform descriptions of the circuits and their behaviours as well as clear and short proofs. An application to a non trivial circuit is given as an illustration.

1 Introduction

Co-induction is a powerful tool for dealing with infinite structures. It is especially well suited to prove properties about circuits where one has to cope with infinitely long temporal sequences. This work presents a general methodology to specifying and proving synchronous sequential circuits in the Calculus of Inductive Constructions (enriched with Co-inductive types) implemented in the Coq proof assistant [1].

It is a continuation of [5], where we made heavy use of dependent types. We go deeply into this direction, introducing dependent types systematically whenever this leads to more precise and reliable specifications. But the main point we focus on in this paper is the use of Coq co-inductive types to implement the notion of time in the representation of clocked circuits. The history of the values carried by the wires of such a device, all along a discrete scale of time, can be encoded very naturally by an infinite list (of co-inductive type *Stream* in Coq). In that way a register, for example, is a function *consing* its initial state value to its input stream. These considerations led us to represent the structure and the behaviour mathematical descriptions by means of Mealy (or Moore) machines. As a matter of fact, these automata, given an initial state and in response to an infinite sequence of inputs, compute an infinite sequence of outputs. They can be implemented straightforwardly in Coq as greatest fixpoints, by means of parameterized co-recursive definitions. They also provide a uniform representation, both for structures and behaviours. In addition, the set of Mealy automata can be enriched with algebraic interconnection rules, and this is particularly valuable in the framework of a modular and hierarchical approach of the verification process.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 91{108, 1999.

Therefore, establishing the correctness of a circuit amounts to proving co-inductively the equivalence between the output stream of its structure and the infinite sequence of outputs computed by its behavioural Mealy machine. One of the main advantages of this methodology, is that the combinatorial part of the proof process is clearly separated from the temporal axe which is treated in a unique co-inductive theorem setting a general proof schema. Any sequential circuit can be verified by applying this theorem after having proved that its hypotheses are satisfied, which is essentially combinatorial.

The implementation in Coq of the automata theory supplies a toolbox which is highly generic. We have used it to verify an important part of the Fairisle ATM Switch Fabric, a real circuit designed, built, and used at the University of Cambridge [11] [10].

The paper is organized as follows. Section 2 presents a brief overview of Coq. Section 3 is dedicated to the description of a generic toolbox implementing the automata theory. Then, in section 4, we present an application of our methodology to the ATM Switch Fabric. Finally, in the last section, we compare our study to other related work.

2 An Overview of Coq

The Coq system [1] is based on the Calculus of Constructions [4] [3] enriched with inductive [14] and co-inductive definitions [9]. Coq's logic is a higher order constructive logic which relies on the Curry-Howard isomorphism and which makes both objects and propositions to be terms of the Lamba-Calculus. The rules for constructing terms are as follows :

- identifiers refer to defined constants or to variables declared in the context,
- $(A\ B)$ denotes the application of a functional object A to B,
- $[x : A]B$ abstracts the variable x of type A in term B in order to construct a functional object, that is generally written $\lambda x \in A.B$ in the literature,
- $(x : A)B$ as a term of type Set corresponds to a product $\prod_{x \in A} B$ of a family of sets B indexed on A. As a term of type Prop, it corresponds to $\forall x \in A\ B$. If x does not occur in B, $A \rightarrow B$ is a short notation which represents either the set of all the functions from A to B or a logical implication.

The system automatically generates the reasoning principles related to each inductive or co-inductive type defined by the user.

As an illustration, let us give some basic definitions that are useful for our development. An infinite list is specified by means of the co-inductive generic type *Stream* with one constructor *Cons* :

```
Variable A:Set.
CoInductive Stream : Set := Cons : A -> Stream -> Stream.
```

The accessors of the type are the two following functions which are defined by cases on the structure of a stream l.

```
Definition Head: Stream -> A := [l] Cases l of (Cons hd _) => hd end.

Definition Tail: Stream -> Stream:= [l] Cases l of (Cons _ tl) => tl end.
```

The co-inductive definition of *EqS* below, expresses that two streams l and l^0 are equivalent iff their heads are equal and their tails are equivalent.

```
CoInductive EqS: Stream -> Stream -> Prop:= eqS: (l, l′ : Stream)
                                                 (Head l) = (Head l′) ->
                                                 (EqS(Tail l((Tail l′))->
                                                 (EqS l l′).
```

Moreover, these definitions can be put inside a section. This is done by writing them between the two declarations :

```
Section Infinite_List.
 ...
End Infinite_List.
```

The advantage is that outside the section, the definitions will be parameterized by the type A of the elements of the streams. The mapping of a given function *f* on two streams *l* and *l'* is co-recursively defined as follows :

```
CoFixpoint Map2 : (A, B, C : Set)
                  (A-> B-> C) -> (Stream A) -> (Stream B) ->(Stream C) :=
   [A, B, f, l, l′]
   (Cons (f (Head l) (Head l′)) (Map2 f (Tail l) (Tail l′))).
```

To be syntactically correct, a co-recursive definition must satisfy guard conditions [9]. In fact, such a declaration is accepted if and only if the recursive occurrences of the variable bounded by this declaration (here the variable $Map2$), are located just under a constructor of the co-inductive type under consideration (here the constructor $Cons$).

Therefore, given two sets *A* and *B*, we can specify the function $Prod$, which builds the stream of the pairs, element by element, of two streams of type *(Stream A)* and *(Stream B)* respectively. $Prod$ is the result of the application of *Map2* to the function *(pair A B)*, where *pair* is the constructor of the cartesian product *A*B* :

```
  Definition Prod := [A, B : Set] (Map2 (pair A B)).
```

After this brief presentation, we can tackle the implementation of the theory of automata.

3 The Mealy Automata Coq Library

3.1 Specification of Mealy Automata

Definition 1 A Mealy automaton is a 5-uple $(I; O; S; Trans; Out)$ where I, O and S are respectively the input set, the output set and the state set.

$Trans : I \times S \rightarrow S$ is called the *transition function* and $Out : I \times S \rightarrow O$ is called the *output function* (see fig. 1).

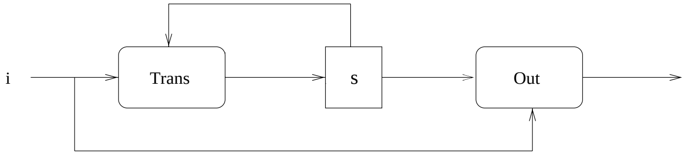


Fig. 1. A Mealy automaton

Such an automaton is defined in Coq by the declaration of 5 variables parameterizing a section :

```
Variables I,O,S : Set.
Variable Trans : I -> S -> S.
Variable Out : I -> S -> O.
```

Let us notice that the functions $Trans$ and Out have been curryfied. Given an initial state s, the Mealy machine computes an infinite output sequence in response to an infinite input sequence. This output stream is computed by the following co-recursive function :

```
CoFixpoint Mealy : (Stream I) -> S -> (Stream O) := [inp, s]
   (Cons (Out (Head inp) s) (Mealy (Tail inp) (Trans (Head inp) s))).
```

The first element of the output stream is the result of the application of the output fonction Out to the first input (that is the head of the input stream inp) and to the initial state s. The tail of the output stream is then computed by a recursive call to $Mealy$ on the tail of the input stream and the new state. This new state is given by the function $Trans$, applied to the first input and the initial state.

The stream of all the successive states from the initial one *s* can be obtained similarly :

```
CoFixpoint States : (Stream I) -> S -> (Stream S) := [inp, s]
   (Cons s  (States (Tail inp) (Trans (Head inp) s))).
```

Once the generic data type is defined, we present the inter-connection rules which express how a complex machine can be decomposed into simpler ones.

3.2 Modularity

Three inter-connection rules are defined on the set of automata. They represent the parallel composition, the sequential composition and the feedback composition of synchronous sequential devices. We shall only develop the first one, as an illustration of the algebraic aspect of the theory.

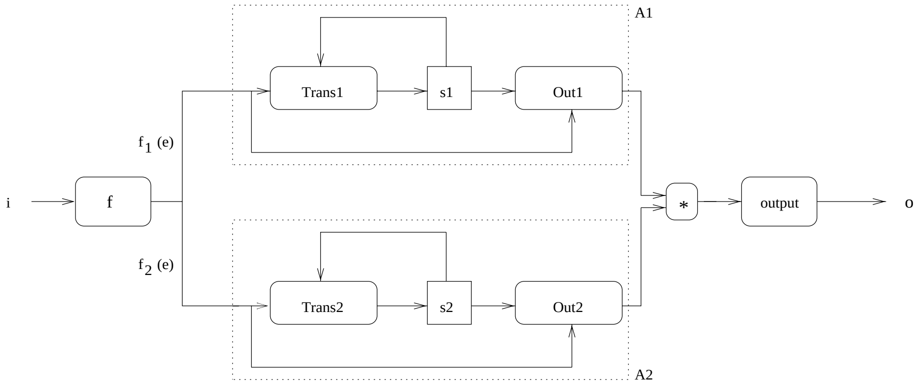


Fig. 2. Parallel Composition of two Mealy Automata

The parallel composition of two Mealy automata $A1$ and $A2$ is informally described in fig. 2. The two objects, on each side of the schema, need comments :

- $f = (f_1; f_2)$ builds from the current input i the pair of inputs $(f_1(i); f_2(i))$ for $A1$ and $A2$.

- $output$ computes the global output from the outputs of $A1$ and $A2$.

This can be implemented in Coq in the following way :

```
Variables I1, I2, O1, O2, S1, S2, I, O : Set.
Variable Trans1 : I1 -> S1 -> S1.  Variable Trans2 : I2 -> S2 -> S2.
Variable Out1 : I1 -> S1 -> O1.    Variable Out2 : I2 -> S2 -> O2.
Variable f : I -> I1*I2.           Variable output : O1*O2 -> O.
Local A1 := (Mealy Trans1 Out1).   Local A2 := (Mealy Trans2 Out2).

Definition parallel : (Stream I) -> S1 -> S2 := [inp, s1, s2]
  (Map output (Prod (A1 (Map Fst (Map f inp)) s1)
                    (A2 (Map Snd (Map f inp)) s2))).
```

In the last definition, $s1$ and $s2$ are the initial states of $A1$ and $A2$. The input of $A1$ is obtained by mapping the first projection Fst on the stream resulting from the mapping of the function f on the global stream inp. Then $(A1\ (Map\ Fst\ (Map\ f\ inp))\ s1)$ is the output stream of $A1$. That of $A2$ is defined similarly. Finally, the parallel composition output stream is obtained by mapping the function $output$ on the product of the output streams of $A1$ and $A2$.

This parallel composition is not an automaton. But it can be shown that it is equivalent to a Mealy automaton called PC in the following sense : if a certain relation holds on the initial states, in response to two equivalent input streams, the output streams for parallel composition and PC are equivalent. The state type of PC is the product of the state type of $A1$ and the state type of $A2$. Its transition function and its output function are respectively defined by :

$$
\begin{aligned}
&8i\ 2\ I;8s_1\ 2\ S_1;8s_2\ 2\ S_2\\
&\qquad Trans\_PC(i;(s_1;s_2)) = (Trans1(f_1(i);s_1);Trans2(f_2(i);s_2)),\\
&\qquad Out\_PC(i;(s_1;s_2)) = (output(Out1(f_1(i);s_1);Out2(f_2(i);s_2))).
\end{aligned}
$$

These definitions are translated in Coq straightforwardly. Let us now prove the equivalence between the parallel composition and this automaton PC.

```
Definition PC:=(Mealy Trans_PC Out_PC).
Lemma Equiv_parallel_PC : (s1 : S1) (s2 : S2) (inp : (Stream I))
                          (EqS (parallel inp s1 s2) (PC inp (s1,s2))).
```

A call to the tactic *Cofix* points to a proof following the co-induction principle associated with the co-inductive definition of *EqS*. It means that the resulting proof term will be co-recursively defined. Practically, it introduces in the context the co-induction hypothesis which is the goal to be proved. An application of this hypothesis stands for a recursive call in the construction of the proof.

```
   Equiv_parallel_PC < Cofix.
1 subgoal
Equiv_parallel_PC : (s1 : S1)      (co-induction hypothesis)
                    (s2 : S2)
                    (inp : (Stream I))
                    (EqS (parallel i s1 s2) (PC i (s1,s2)))
  ============================
(s1:S1)(s2:S2)(inp:(Stream I))(EqS (parallel inp s1 s2) (PC inp (s1,s2)))
```

Of course, at this point, this hypothesis may not be used to prove the goal (petitio principii). Indeed, the resulting proof term would be rejected by the system, as syntactically incorrect since it does not satisfy the required guard conditions. Therefore, we use the tactic *Intros* to introduce in the context the hypotheses $s1$, $s2$ and inp and then, by applying the constructor eqS of EqS, we split the goal into two new subgoals which mean that we have now to prove the equality of the heads and the equivalence of the tails.

```
Equiv_parallel_PC < (Intros s1 s2 inp ; Apply eqS).
2 subgoals
  Equiv_parallel_PC : (s1:S1)(s2:S2)(inp:(Stream I))
                      (EqS (parallel inp s1 s2) (PC inp (s1,s2)))
  s1 : S1
  s2 : S2
  inp : (Stream I)
  ============================
   (Head (parallel inp s1 s2))=(Head (PC inp (s1,s2)))


subgoal 2 is:
 (EqS (Tail (parallel inp s1 s2)) (Tail (PC inp (s1,s2))))
```

The first subgoal is automatically resolved, after having been simplified by -reduction.

```
Equiv_parallel_PC < (Simpl; Auto).
1 subgoal
  Equiv_parallel_PC : (s1:S1)(s2:S2)(inp:(Stream I))
                      (EqS (parallel inp s1 s2) (PC inp (s1,s2)))
  s1 : S1
  s2 : S2
  inp : (Stream I)
  ============================
   (EqS (Tail (parallel inp s1 s2)) (Tail (PC inp (s1,s2))))
```

The equivalence of the tails is now automatically established by an application of the co-induction hypothesis (tactic *Trivial*) after unfolding the definition of *parallel* and simplifying the resulting term.

```
Equiv_parallel_PC < (Unfold parallel in Equiv_parallel_PC;Simpl;Trivial).
Subtree proved!
```

For clarity, we give in a slightly simplified syntax the co-recursive proof term generated by this session :

```
CoFixpoint Equiv_parallel_PC: (s1 : S1) (s2 : S2) (inp : (Stream I))
                              (EqS(parallel inp s1 s2) (PC inp(s1,s2))):=
[s1 : S1] [s2 : S2] [inp : (Stream I)]
   (eqS (refl_equal O (output ((Out1 (Fst (f (Head inp))) s1),
                               (Out2 (Snd (f (Head inp))) s2))))
        (Equiv_parallel_PC (Trans1 (Fst (f (Head inp))) s1)
                           (Trans2 (Snd (f (Head inp))) s2)
                           (Tail inp))).
```

We see that, in the declaration above, the recursive call to *Equiv_para-llel_PC* occurs under the constructor *eqS*.

Moreover, two automata are said equivalent if their outputs are equivalent streams whenever their inputs are equivalent streams. We proved an important fact, namely that the equivalence of automata is a congruence for the composition rules. For example, if A_1 and A_2 are two equivalent automata and B_1 and B_2 are equivalent as well, then the parallel composition of A_1 and A_2 is an automaton equivalent to the parallel composition of B_1 and B_2.

3.3 Proof Schema for Circuit Correctness

Proving that a circuit is correct amounts to proving that, under certain conditions, the output stream of the structural automaton and that of the behavioural automaton are equivalent. We present in this section a generic lemma, all our correctness proofs rely on. It is in fact a kind of pre-established proof schema which handles the main temporal aspects of these proofs. Let us first introduce some specific notions.

In the following, we consider two Mealy automata :

$$A1 = (I;\ O;\ S_1;\ Trans\_1;\ Out\_1) \text{ and } A2 = (I;\ O;\ S_2;\ Trans\_2;\ Out\_2)$$

that have the same input set and the same output set.

Invariant Given p streams, a relation which holds for all p-tuples of elements at the same rank, is called an invariant for these p streams.

Let us specify this notion in Coq for 3 streams on the sets I, S_1 and S_2.

```
CoInductive Inv [P : I -> S1 -> S2 -> Prop] :
                (Stream I) -> (Stream S1) -> (Stream S2) -> Prop :=
  C_Inv : (inp : (Stream I))(st1 : (Stream S1))(st2 : (Stream S2))
          (P (Head inp) (Head st1) (Head st2)) ->
          (Inv P (Tail inp) (Tail st1) (Tail st2)) ->
          (Inv P inp st1 st2).
```

Now,$(Inv\,P\ inp\,st1\,st2)$ means that P is an invariant for the triplet $(inp;st1;st2)$.

Invariant state relation Let R be a relation on $S_1\quad S_2$ and P a relation on $I\quad S_1\quad S_2$. R is invariant under P for the automata $A1$ and $A2$, iff :

$$8i\ 2\ I;8s_1\ 2\ S_1;8s_2\ 2\ S_2;$$
$$(P\ (i;\ s_1;\ s_2)\ \hat{}\ R\ (s_1;\ s_2))\ \ )\ \ R\ (Trans\_1\ (i;\ s_1);Trans\_2\ (i;\ s_2)).$$

This is translated in Coq by :

```
Definition Inv_under:=[P: I -> S1 -> S2 -> Prop] [R : S1 -> S2 -> Prop]
   (i : I) (s1 : S1) (s2 : S2)
   (P i s1 s2) -> (R s1 s2) -> (R (Trans1 i s1) (Trans2 i s2)).
```

Output relation A relation on the states of two automata is an output relation if it is strong enough to induce the equality of the outputs.

```
Definition Output_rel := [R: S1 -> S2 -> Prop]
                         (i : I) (s1 : S1) (s2 : S2)
                         (R s1 s2) -> (Out1 i s1) = (Out2 i s2).
```

We can now set the equivalence lemma :

```
Lemma Equiv_2_Mealy :
   (P : I -> S1 -> S2 -> Prop) (R : S1 -> S2 -> Prop)
   (Output_rel R) -> (Inv_under P R) -> (R s1 s2) ->
   (inp : (Stream I)) (s1 : S1) (s2 : S2)
   (Inv P inp (States Trans1 Out1 inp s1) (States Trans2 Out2 inp s2)) ->
   (EqS (A1 inp s1) (A2 inp s2)).
```

in other words if R is an output relation invariant under P that holds for the initial states, if P is an invariant for the common input stream and the state streams of each automata, then the two output streams are equivalent. The proof of this lemma is done by co-induction.

4 Application to the Certification of a True Circuit

4.1 The 4 by 4 Switching Element

Designed and implemented at Cambridge University by the Systems Research Group, the Fairisle 4 by 4 Switch Fabric is an experimental local area network based on Asynchronous Transfer Mode (ATM).

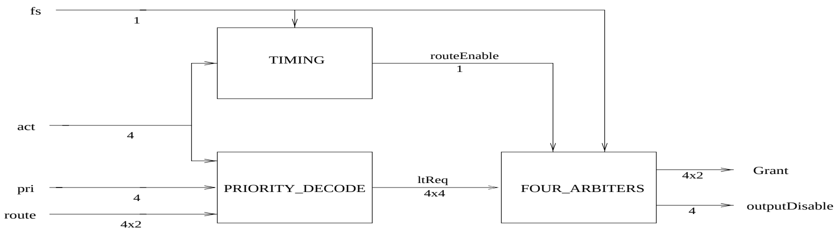


Fig. 3. Arbitration Unit

The switching element is the heart of the 4 by 4 Switch Fabric, connecting 4 input ports to 4 output ports. Its main role is performing switching of data from input ports to output ports and arbitrating data clashes according to the output port requests made by the input ports. We focus here on its *Arbitration* unit, which is its most significant part, as far as specification and verification are concerned. This unit decodes requests from input ports and priorities between data to be sent, and then it performs arbitration. It is the interconnection of 3 modules (fig.3) :

- *FOUR_ARBITERS* which performs the arbitration for all output ports, following the Round Robin algorithm,
- *TIMING* which determines when the arbitration process can be triggered,
- *PRIORITY_DECODE* which decodes the requests and filters them according to their priority. Its structure is essentially combinatorial.

As an illustration of section 3.3, we present in detail the verification of *TIMING*, which is rather simple and significant enough to illustrate the proof of equivalence between a structural automaton and a behavioural automaton. Then, we shall present how *ARBITRATION* is verified by joining together the various correctness results of its sub-modules. This will illustrate not only the hierarchical aspect of our approach, but also that the real objects we have to handle are in general much more complex than those presented on the example of *TIMING*

4.2 Verification of the Basic Unit Timing

The unit *TIMING* can be specified and proved directly, that is without any decomposition. It is essentially composed of a combinatorial part connected to two registers as shown in fig.4.

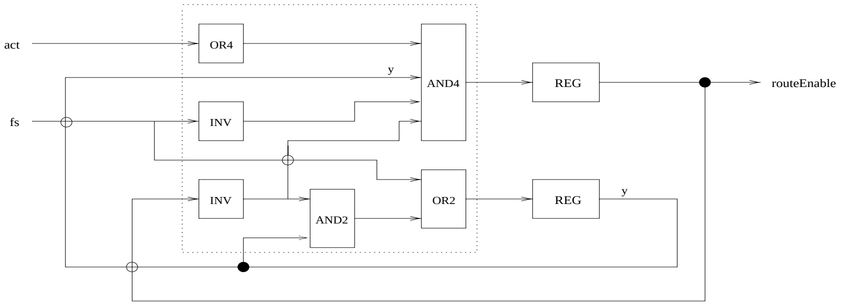


Fig. 4. Timing Unit

Structure The structure of *TIMING* corresponds exactly to a Mealy automaton. Its transition function represents the boxed combinatorial part in fig.4. The state is the pair of the two register values. The definitions below give some examples of logical gates encoding. The function *neg* and *andb* are functions defined in a library dedicated to the booleans.

```
Local I := bool * (d_list bool four).
Local O := bool.
Definition S := (d_list bool two).
Definition INV := neg.
Definition AND4 := [a, b, c, d : bool] (andb a (andb b (andb c d))).

Definition Timing : I -> S -> S :=
   [i, s0] let (fs, act) = i in
          (List2 (AND4 (OR4 act) (Snd2 s0) (INV fs) (INV (Fst2 s0)))
                 (OR2 fs (AND2 (INV (Fst2 s0)) (Snd2 s0)))).

Definition Out_Struct_Timing I -> S -> O := [_ , l]  (Fst2 l).
Definition Structure_TIMING := (Mealy Timing Out_Struct_Timing).
```

The keyword *Local* introduces declarations which are local to the current section. We have given the instanciation of the type *I* as an example of a real data type. The input is composed of two signals (*fs*, *act*) of 1 bit and 4 bits respectively (see fig.4). Therefore, *fs* is coded by a term of type *bool*, but *act* is a bit more complex. It is represented by a list of four booleans of type *(d_list bool four)*. This type is a dependent type : it depends on the length (here the term *four*) of the lists under consideration. We do not go into details about dependent types (see [5]), but we recall that, although they are quite tricky to handle, they provide much more precise specifications.

Behaviour The behaviour is presented in fig.5. The output is a boolean value that indicates when the arbitration can be triggered. This output is false in general. When the frame start signal *fs* goes high, the device waits until one of the four values carried by *act* is true. In that case the output takes the value *true* during one time unit and then it goes low again. The type of the states is defined by :

```
Inductive S_beh: Set := START_t: S_beh | WAIT_t: S_beh | ROUTE_t: S_beh.
```

The transition function *trans_T* and the output function *out_T* are simple and defined by cases. The automaton *Behaviour_TIMING* is obtained as usual by an instanciation of *Mealy*.

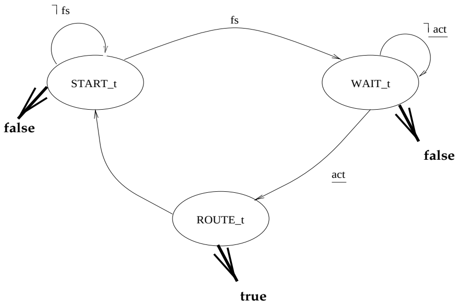


Fig. 5. Timing Behaviour

Proof of Equivalence All the notions we use in this paragraph have been introduced in section 3.3. To prove the equivalence between *Behaviour_TIMING* and *Structure_TIMING* we apply the lemma *Equiv_2_Mealy* of our toolbox. For that, we have to define a relation between the state *l* of the structure and the state *s* of the behaviour. The relation *R_Timing* expresses that the first register value (first element of *l*) equals the output of the behaviour in the state *s*, that insures that the relation is an *output relation*. It also expresses constraints between *s* and the second register of the structure.

```
Definition R_Timing S_beh -> S -> Prop :=
[s, l]((i:I)(Fst2 l)=(Out_T i s)) /\ (s = START_t /\ (Snd2 l) = false \/
                                      s = WAIT_t  /\ (Snd2 l) = true  \/
                                      s = ROUTE_t /\ (Snd2 l) = true).
```

No additional hypothesis is needed to prove that *R_Timing* is an invariant relation on the state streams, that means that it is invariant under the *True* constant predicate on the streams, which is obviously an invariant. Let us call it *Cst_True*. After having proved that *R_Timing* is a relation invariant under *Cst_True*, the required correctness result is obtained as a simple instanciation of the lemma *Equiv_2_Mealy* as shown in the trace below.

```
Lemma Inv_relation_T: (Inv_under Trans_T Timing Cst_True R_Timing).
...
Lemma Correct_TIMING: (i: (Stream I)) (l: S) (s: S_beh)
                      (R_Timing s l)->
                      (EqS (Behaviour_TIMING i s) (Structure_TIMING i l)).

  Intros i l s HR.
  (Unfold Behaviour_TIMING; Unfold Structure_TIMING).
  Apply (Equiv_2_Mealy Inv_relation_T Output_rel_T Inv_Cst_True HR).
  Save.
```

In the proof above, *Output_rel_T* and *Inv_Cst_True* stand for the proof that *R_Timing* is an output relation and that *Cst_True* is an invariant. This example is rather simple but it clearly shows that the combinatorial part of the proof and the temporal one are separated. The former essentially consists in proving that *R_Timing* is an invariant, which is essentially a proof by cases. The latter has been already done when we proved the lemma *Equiv_2_Mealy*.

4.3 Hierarchical and Modular Aspects

Let us now illustrate the hierarchical modularity of our approach on the specification and the verification of an interconnection of several modules, namely the Arbitration unit (fig.3). Its structure is described in fig.6.

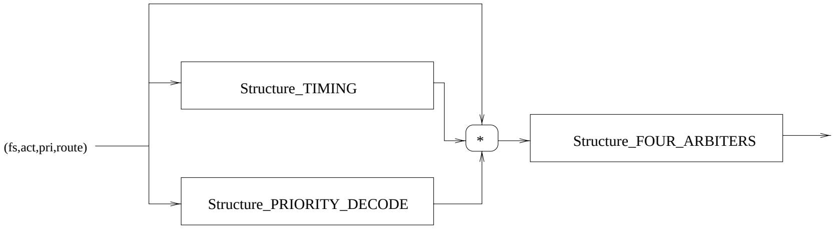


Fig. 6. The Arbitration Structure as an Interconnection of Automata

Structural Description The structural description is given in several steps (fig.7, 8, 9), by using the parallel and sequential composition rules (*PC* and *SC*) on automata presented in paragraph 3.2.

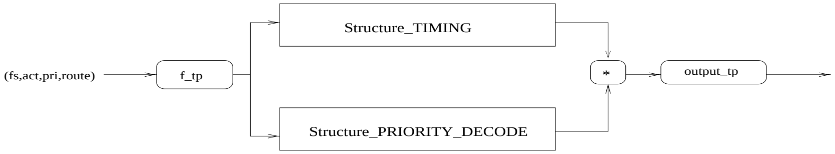


Fig. 7. *Structure_TIMINGPDECODE*

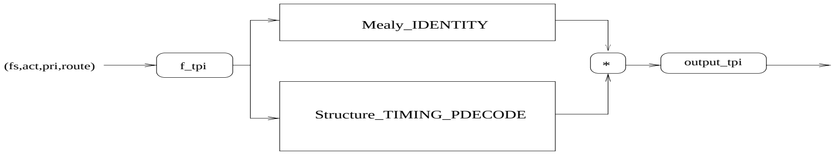


Fig. 8. *Structure_TIMINGPDECODE_ID*

We only give the Coq code of the final definition representing the structure of *ARBITRATION* as a sequential composition of two intermediate automata.

```
Local I :=    bool * (d_list bool four) * (d_list bool four) *
             (d_list (d_list bool two) four).

Definition Structure_ARBITRATION : (Stream I) -> S -> (Stream O) :=
   (SC  TransPC_TmgPdecode_id  Trans_struct_four_arbiters
        OutPC_TmgPdecode_id    Out_struct_four_arbiters  ).
```

The term *SC* builds the Mealy automaton equivalent to the sequential composition of the automata *Structure_TIMINGPDECODE_ID* and *Structure_FOUR_ARBITERS* (fig.9).

Behavioural Description The behaviour of *Arbitration* was initially given in natural language by the designers. Several formal descriptions can be derived. For example, in HOL, Curzon uses classical timing diagrams (waveforms) whereas Tahar, in MDG, specifies it by means of Abstract State Machines (ASM) the states of which are located according to two temporal axis [16]. Our description is more abstract, more compact and closer to the informal statement. We represent it, as usual, by a Mealy automaton which is described in fig.10. It is particularly small (only 5 states). This comes from the fact that a great deal of information is expressed by the states themselves, the output function and the transition function.

The input has the same type *I* as in the structural description. The current input is then *(fs, act, pri, route)*. The states are 4-tuples consisting of :

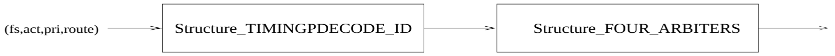


Fig. 9. *Structure_ARBITRATION*

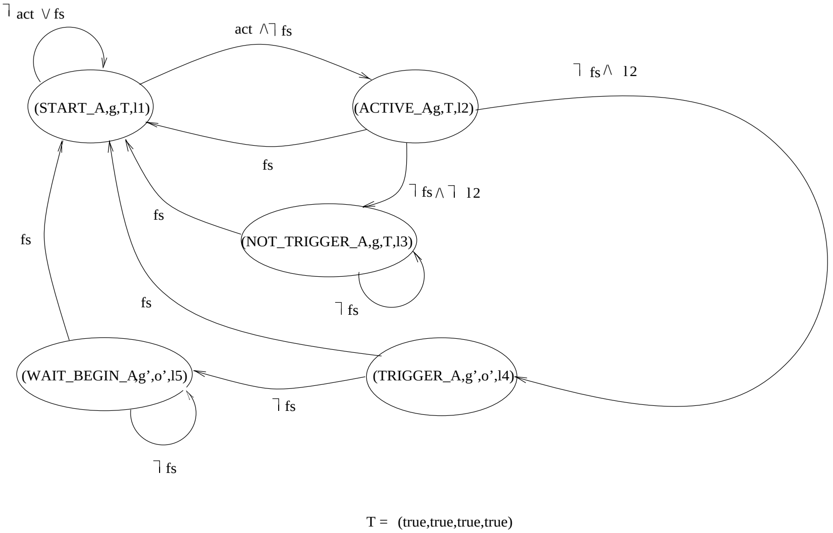


Fig. 10. Arbitration Behaviour

{ a label (START_A for instance),
{ a list g of 4 pairs of booleans, each of them being the binary code of the last input port that gained access to the output port corresponding to its rank in the list g,
{ a list o of 4 booleans indicating if the information of same rank in the list g above is up to date (an element of g can be out of date if the corresponding output port has not been requested at the previous cycle),
{ the current requests l. It is a list of 4 elements (one for each output port). Each of these elements is itself a list of 4 booleans (one for each input port) indicating if the input port actually requests the corresponding output port and if it has a high level of priority or not.

The transition function computes the new state from the information carried by the current state and the current input. Hence, it is quite complex. Let us explain how g and o are updated. This is done my means of an arbitration process. For each output port, it first computes the last port (say $last$) that got access to this output port. Then, it calls the $RoundRobin\,(4; l; last)$ where $RoundRobin$ is described as follows :

```
RoundRobin (n; l; last)  =  if  n = 0  then
                                 (last; true)
                               else
                                 let succ = (last + 1) mod 4  in
                                   if succ 2 l then
                                      (succ; false)
                                   else
                                      RoundRobin (n − 1; l; succ)
```

The list of the first components returned by the 4 calls (one for each output port) to *RoundRobin* constitutes the new value of *g* and the list of the second components, that of *o*. We do not go into details about the new requests that are computed by decoding and filtering the current input.

For lack of space, we do not give the Coq code of this transition function. Let us just make clear that, at each step, it describes non trivial intermediate functions for arbitrating, filtering, decoding. This points out an essential feature of our specification : due to the high level of abstraction of the Coq specification language, we can handle automata which have few states but which carry a lot of information. This allows us to avoid combinatorial explosions and leads to short proofs (few cases have to be considered). The automaton outputs the list product of *g* and *o*. Let us give the states type *S* :

```
Inductive label : Set := WAIT_BEGIN_A : label |
                         TRIGGER_A : label |
                         START_A : label |
                         ACTIVE_A : label |
                         NOT_TRIGGER_A : label.
Definition S : Set :=   label
                      * (d_list bool*bool four)
                      * (d_list bool four)
                      * (d_list (d_list bool four) four).

Definition Trans_Arbitration : I -> S -> S := ...
Definition Out_Arbitration : I -> S -> O := ...

Definition Behaviour_ARBITRATION:=
                                (Mealy Trans_Arbitration Out_Arbitration).
```

The Proof of Correctness: An Outline The proof of correctness follows from the verification of the modules that compose the *Arbitration* unit. We perform it in several steps, hierarchically.

(1) We build behavioural automata for *TIMING*, *FOUR_ARBITERS*, and *PRIORITY_DECODE*. We prove that these three automata are equivalent to the three corresponding structural automata.

(2) We interconnect the structural automata and we get the global structural automaton called *Structure_ARBITRATION*.

(3) In the same way, we interconnect the three behavioural automata (1) and we get an automaton called *Composed_Behaviours*.

(4) We show, from (1) and by applying the lemmas stating that the equivalence of automata is a congruence for the composition rules, that *Composed_Behaviours* and *Structure_ARBITRATION* are equivalent.

(5) We prove that *Composed_Behaviours* is equivalent to the expected behaviour, namely *Behaviour_ARBITRATION*. This is the essential part of the global proof and is much simpler than proving directly the equivalence between the structure and the behaviour. As a matter of fact, *Composed_Behaviours* is more abstract than *Structure_ARBITRATION* which takes into account all the details of the implementation.

(6) This final result, namely the equivalence of *Behaviour_ARBITRATION* and *Structure_ARBITRATION*, is obtained easily from (4) and (5) by using the transitivity of the equivalence on the *Streams*.

Let us point out that the lemma *Equiv_2_Mealy* of our toolbox (section 3.3) has been applied several times (3 times in (1) and once in (5)). Here is the final lemma :

```
Lemma Correct_ARBITRATION : (i : (Stream I))
(EqS (Behaviour_ARBITRATION i
            (WAIT_BEGIN_A,
            ((List4 g11_0 g12_0 g21_0 g22_0),(l4_ffff,p_0))))
      (Structure_ARBITRATION i
            ((IDENTITY,
            (l2_ff,p_0)),
            ((((pdt_List2 g11_0),false),((pdt_List2 g12_0),false)),
            (((pdt_List2 g21_0),false),((pdt_List2 g22_0),false)))))).
```

Moreover, it is worth noticing that the *FOUR_ARBITERS* unit is itself composed of 4 sub-units and that its verification requires again a modular verification process.

5 Conclusion

We have presented in this paper a methodology, entirely implemented in Coq, for specifying and verifying synchronous sequential circuits. The starting point is a uniform description of the structure and the expected behaviour of circuits by means of Mealy automata. The points of our work that must be emphasized are the following :

- The use of Coq dependent types : although they are tricky to use (to our point of view), they provide very precise and reliable specifications.
- The use of Coq co-inductive types : we could obtain a clear and natural modelling of the history of the wires in a circuit without introducing any temporal parameter.

- Reasoning by co-induction : we could capture once and for all in one generic lemma most of the temporal aspects of the proof processes. In each specific case, only combinatorial parts need to be developed.
- The generic library on automata that has been developed : it can be reused in every particular case.
- The feasibility of our approach, that has been demonstrated on the example of a real non trivial circuit. Our whole development (including the generic tools) takes approximatively 4,000 lines.
- The hierarchical and modular approach : not only does this lead to clearer and easier proof processes but also this allows us to use, in a complex verification process, correctness results related to pre-proven components.
- The small size of the automata we define : at most 5 states for the circuit under consideration. This comes from the complex structure of the states that carry a lot of information. Therefore the proofs by cases are short but they make use of high level transition functions on rich data types.

A great deal of work has been performed in the field of hardware verification using proof assistants. Let us mention those closest to ours. In [15] Paulin-Mohring gave a proof of a multiplier, using a codification of streams in type theory, in a former version of Coq in which co-inductive types had not been implemented yet. In [13] a formalization of streams in PVS uses parameterized types to circumvent the absence of co-induction in PVS. Its aim is to verify a synchronous fault-tolerant circuit. More recently, Cachera [2] has shown how to use PVS to verify arithmetic circuits described in Haskell.

The ATM Switch Fabric has been (and still is) widely used as a benchmark in the hardware community. Let us cite Curzon[7] [6] who has specified and proved this circuit in HOL. His study has been a helpful starting point for our investigations despite his approach is completely different in the sense that he specifies the structures as relations that are recursive on a time parameter and he represents the behaviours by timing diagrams. He does not obtain parameterized libraries but rather libraries related to specific pieces of hardware. As most of his proofs are inductive, each proof requires at least one particular induction, and sometimes several nested inductions with various base cases. This has to be contrasted with our unique generic temporal lemma. Tahar in [18] proved the Fabric using MDG (Multiway Decision Graphs). He handles bigger automata and his proof is more automatic. However it is not reusable. Other approaches on the Fabric propose abstraction processes in order to alleviate the proof process [12] [8]. Several comparisons have been studied in this field [17] [16].

References

[1] B. Barras and al. The Coq Proof Assistant Reference Manual : Version 6.1. Technical Report 0203, INRIA-Rocquencourt, CNRS-ENS Lyon, France, Dec. 1997.

[2] D. Cachera. Verification of Arithmetic Circuits using a Functional Language and PVS. Technical Report 97-48, ENS-Lyon, LIP, Dec. 1997.

[3] T. Coquand. *Une Theorie des Constructions*. PhD thesis, Universite Paris 7, Janvier 1989.

[4] T. Coquand and G. Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. *EUROCAL 85, Linz Springer-Verlag LNCS 203*, 1985.

[5] S. Coupet-Grimal and L. Jakubiec. Coq and Hardware Verification : a Case Study. In J. G. J. von Wright and J. Harrison, editors, *TPHOLs'96*, LCNS 1125, pages 125{139, Turku (Finlande), 27-30th August 1996. Springer-Verlag.

[6] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element. Technical Report 329, University of Cambridge, Mar. 1994.

[7] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element: an Overview. Technical Report 328, University of Cambridge, Mar. 1994.

[8] E. Garcez and W. Rosenstiel. The Verification of an ATM Switching Fabric using the HSIS Tool. *IX Brazilian Symposium on the Design of Integrated Circuits*, 1996.

[9] E. Gimenez. *Un calcul de constructions infinies et son application a la verification de systemes communicants*. PhD thesis, Ecole Normale Superieure de Lyon, 1996.

[10] I. Leslie and D. McAuley. Fairisle : A General Topology ATM LAN. *http://www.cl.cam.ac.uk/Research/SRG/fairpap.html*, Dec. 1990.

[11] I. Leslie and D. McAuley. Fairisle : An ATM Network for the Local Area. *ACM Communication Review*, 4(19):327{336, September 1991.

[12] J. Lu and S. Tahar. Practical Approaches to the Automatic Verification of an ATM Switch using VIS. In *IEEE 8th Great Lakes Symposium on VLSI (GLS-VLSI'98)*, pages 368{373, Lafayette, Louisiana, USA, Feb. 1998. IEEE Computer Society Press.

[13] P. S. Miner and S. D. Johnson. Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit. In *Designing Correct Circuits*. Bastad, 1996.

[14] C. Paulin-Mohring. Inductive Definition in the System Coq : Rules and Properties. *Typed Lambda Calculi and Applications (Also Research Report 92-49, LIP-ENS Lyon)*, Dec. 1993.

[15] C. Paulin-Mohring. Circuits as Streams in Coq. Verification of a Sequential Multiplier. *Basic Research Action "Types"*, July 1995.

[16] S. Tahar and P. Curzon. A Comparison of MDG and HOL for Hardware Verification. In J. G. J. von Wright and J. Harrison, editors, *TPHOLs'96*, LCNS 1125, pages 415{430, Turku (Finlande), 27-30th August 1996. Springer-Verlag.

[17] S. Tahar, P. Curzon, and Lu. J. Three Approaches to Hardware Verification : HOL, MDG and VIS Compared. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, LCNS 1522, pages 433{450, FMCAD'98, Palo Alto, California, USA, Nov. 1998. Springer-Verlag.

[18] S. Tahar, Z. Zhou, X. Song, E. Cerny, and M. Langevin. Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs. In *Proc. of the Great Lakes Symp. on VLSI*, IEEE Computer Society Press, pages 106{111, Mar. 1996.

Connecting Proof Checkers and Computer Algebra Using *OpenMath*

Olga Caprotti and Arjeh M. Cohen

RIACA, Technical University of Eindhoven
The Netherlands
{olga,amc}@win.tue.nl

Interactive mathematics over the Internet is particularly appealing in that it can take advantage of the wealth of resources available online. In particular, a problem-solving framework integrating computing and proving tools is of great interest. Several modes of advantageous cooperation can be envisioned. Proving is undoubtedly useful to computation for checking assertions and side conditions, whereas computation is useful for shortening the lengths of proofs. Both modes are definitely needed in interactive mathematics. In the interaction we do not want to waste resources by sending or receiving meaningless requests. There *Strong OpenMath* comes in, as it is possible to type-check the well-typedeness, hence the meaningfullness, of the mathematical objects it represents.

Strong OpenMath is a substantial sublanguage of the *OpenMath* standard for the representation of mathematical objects. Started as a language for communicating between computer algebra systems [3], *OpenMath* has evolved and now aims at being a universal interface among systems of computational mathematics [7, 1].

The major novelty of the *OpenMath* language is the introduction of binding objects to express variable binding. For instance, the mathematical function $\lambda x.x + 2$ is represented by the *OpenMath* object

$$\mathtt{binding}(\mathtt{lambda}, x, \mathtt{application}(\mathtt{plus}\ x\ 2)).$$

The *OpenMath* standard does not adopt a specific type system but allows for formally expressing signatures of symbols. By defining and using a Content Dictionary for representing terms of the chosen type system, formal signatures of *OpenMath* symbols are made to correspond to types representable as *OpenMath* binding objects built using this CD. As added bonus, logical properties of an *OpenMath* symbol can also be formally defined as *OpenMath* binding objects and can be included as such in the symbol's definitions.

A standard technique in formal methods is to use formally specified signatures to assign mathematical meaning to the object in which the symbol occurs. By doing this, validation of *OpenMath* objects depends exclusively on the context determined by the CDs and on some type information carried by the objects themselves. Determining a type for an *OpenMath* object corresponds to assigning mathematical meaning to the object. The subclass of *OpenMath* objects for which this can be done is called *Strong OpenMath*.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 109–112, 1999.

Strong OpenMath is the fragment of *OpenMath* that can be used to express formal mathematical objects, so that formal theorems and proofs, understandable to proof checkers, can be unambiguously communicated, as well as the usual mathematical expressions handled by CA systems.

We show this approach using the Calculus of Constructions (CC) and its extensions as starting point for assigning signatures to *OpenMath* symbols. Abstraction and function space type are represented by introducing in a new Content Dictionary, called `cc`, appropriate binding symbols (`lambda`, `PiType`). Additionally, `cc` also provides the symbols for the names of the types of the basic *OpenMath* objects. The Content Dictionary `ecc` adds to the symbols in `cc`, the symbol `Pair` for pairing, `PairProj1` and `PairProj2` for projections and `SigmaType` for the cartesian product. Using `ecc`, terms and types arising in the Extended Calculus of Constructions can be expressed as *Strong OpenMath* objects.

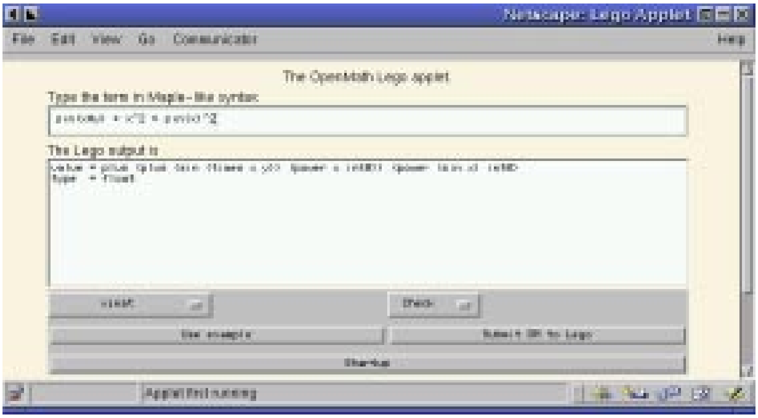
Since the Calculus of Constructions and its extensions have decidable type inference, they are good candidate type systems for proof checkers. Type checking algorithms has been implemented in systems like Lego or COQ [4, 6]. These systems, if *OpenMath* compliant, can provide the functionalities for performing type checks on *OpenMath* objects.

The authors are working on exploiting the type-checking functionalities of COQ and Lego for *Strong OpenMath* through several examples of Java client-server applet. The implementations use the *OpenMath* Java library provided by the PolyMath Development Group [5] and extend it by encoder/decoder classes that act as Phrasebooks for COQ and Lego.

Currently, the Lego Phrasebook transforms any *Strong OpenMath* object into the corresponding ECC term in the language used by Lego. Basic objects are translated into new constants of the appropriate type. Figure 1 is a screenshot of the applet that shows the result of type-checking the expression $\sin(x * y) + x^2 + \sin(x)^2$ in a specific Lego context.

Figure 1 shows that the *OpenMath*-Lego applet takes as input a mathematical expression in Maple syntax. The Maple Phrasebook provided by the *OpenMath* Java library converts this expression into an *OpenMath* object. The Java applet then feeds this object to the Lego Phrasebook for obtaining the corresponding Lego term. This Lego term is then shipped to Lego with a context and a command to be performed on the term. Lego runs on a server machine and communicates to the applet the results of the query it has received. In particular, the query to Lego can be type-checking. If type-checking is successful, then the *Strong OpenMath* object corresponding to the input expression is a meaningful mathematical expression. If type-checking fails, then the resulting error message is displayed and can be caught.

Another version of the applet called *OpenMath* STARS-COQ Applet is based on the same client-server model but in combination with different Phrasebooks and different servers. Figure 2 shows a customized syntax for the input of mathematical expressions via the STARS applet. The corresponding term is rendered



[illegible]

```
value = plus (plus (sin (times x y)) (power x int(2))) (power (sin x) int(2))
type  = float
```

Fig. 1. Maple-Lego Applet

by STARS using *OpenMath* and MathML technology. The term is then transformed to *OpenMath* and sent to COQ for further processing.

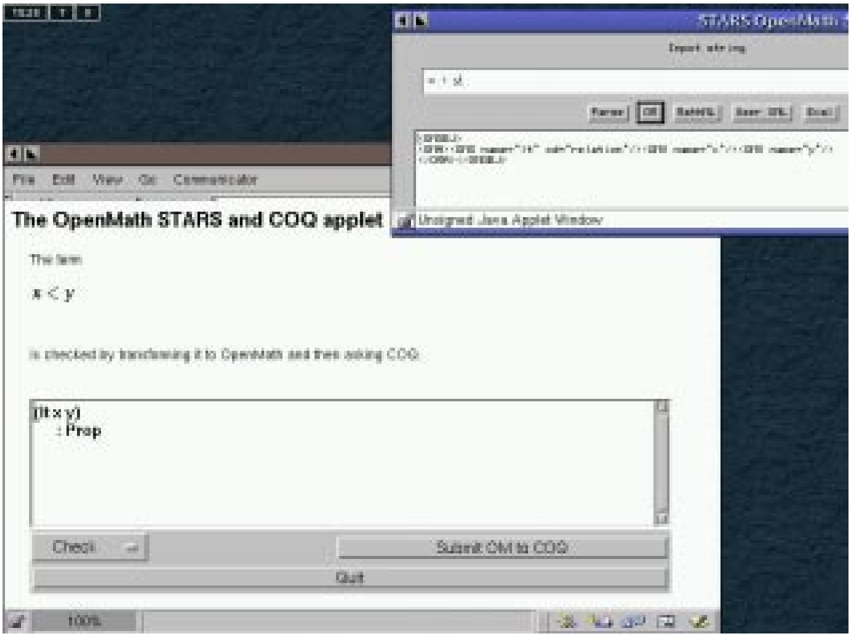
Two particular applications we have in mind are linked to the IDA project, which brought forth "Algebra Interactive". This is interactive first year undergraduate algebra course material for mathematics and computer science students, see [2]. First, user input in a specific input field usually represents a particular type of mathematical object (e.g., an integer, a polynomial over a specified ring and with specified indeterminate, or a permutation group). Before sending the input to a computer algebra server, it is useful to check at the client side whether the input has the right type. This can easily be handled by the present OpenMath-Lego Applet and will become useful when more than one computer algebra engine (presently GAP) will be employed. More complicated and more of a challenge is the second application we have in mind: turning the present static proofs to interactive events. Here research is needed regarding the correspondence of vernacular with formal mathematics and the selection of "interesting" steps from a formal proof.

Of course, this "Algebra Interactive" example is just one of many possibile ways of incorporating mathematical software in an environment using OpenMath.

References

[1] John A. Abbott, Andre van Leeuwen, and A. Strotmann. *OpenMath*: Communicating Mathematical Information between Co-operating Agents in a Knowledge Network. *Journal of Intelligent Systems*, 1998. Special Issue: "Improving the Design of Intelligent Systems: Outstanding Problems and Some Methods for their Solution.".

[2] A. M. Cohen, H. Cuypers, and H. Sterk. *Algebra Interactive, interactive course material*. Number ISBN 3-540-65368-6. SV, 1999.



The OpenMath STARS and COQ applet

The term

$x < y$

is checked by transforming it to OpenMath and then asking COQ

```
(lt x y)
  : Prop
```

Fig. 2. STARS-COQ Applet

[3] S. Dalmas, M. Gaëtano, and S. Watt. An OpenMath 1.0 Implementation. pages 241{248. ACM Press, 1997.

[4] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual*. Department of Compter Science, University of Edinburgh, 1992.

[5] PolyMath OpenMath Development Team. Java openmath library, version 0.5. Available at `http://pdg.cecm.sfu.ca/openmath/lib/`, July 1998.

[6] Projet Coq. *The Coq Proof Assistant: The standard library*, version 6.1 edition. Available at `http://www.ens-lyon.fr/LIP/groupes/coq`.

[7] The OpenMath Consortium. The OpenMath Standard. OpenMath Deliverable 1.3.1a, September 1998. Public at `http://www.nag.co.uk/projects/OpenMath.html`.

A Machine-Checked Theory of Floating Point Arithmetic

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway
Hillsboro, OR 97124, USA

Abstract. Intel is applying formal verification to various pieces of mathematical software used in Merced, the first implementation of the new IA-64 architecture. This paper discusses the development of a generic floating point library giving definitions of the fundamental terms and containing formal proofs of important lemmas. We also briefly describe how this has been used in the verification effort so far.

1 Introduction

IA-64 is a new 64-bit computer architecture jointly developed by Hewlett-Packard and Intel, and the forthcoming Merced chip from Intel will be its first silicon implementation. To avoid some of the limitations of traditional architectures, IA-64 incorporates a unique combination of features, including an instruction format encoding parallelism explicitly, instruction predication, and speculative/advanced loads [4]. Nevertheless, it also offers full upwards-compatibility with IA-32 (x86) code.[1]

IA-64 incorporates a number of floating point operations, the centerpiece of which is the fma (floating point multiply-add or fused multiply-accumulate). This computes $xy + z$ from inputs x, y and z with a single rounding error. Floating point addition and multiplication are just degenerate cases of fma, $1y + z$ and $xy + 0$.[2] On top of the primitives provided by hardware, there is a substantial suite of associated software, e.g. C library functions to approximate transcendental functions.

Intel has embarked on a project to formally verify all Merced's basic mathematical software. The formal verification is being performed in HOL Light, a version of the HOL theorem prover [6]. HOL is an interactive theorem prover in the 'LCF' style, meaning that it encapsulates a small trusted logical core and implements all higher-level inference by (usually automatic) decomposition to these primitives, using arbitrary user programming if necessary.

A common component in all the correctness proofs is a library containing formal definitions of all the main concepts used, and machine-checked proofs of

[1] The worst-case accuracy of the floating-point transcendental functions has actually improved over the current IA-32 chips.

[2] As we will explain later, this is a slight oversimplification.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 113–130, 1999.

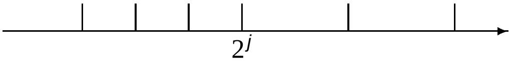
a number of key lemmas. Correctness of the mathematical software starts from the assumption that the underlying hardware floating point operations behave according to the IEEE standard 754 [9] for binary floating point arithmetic. Actually, IEEE-754 doesn't explicitly address `fma` operations, and it leaves underspecified certain significant questions, e.g. NaN propagation and underflow detection. Thus, we not only need to specify the key IEEE concepts but also some details specific to IA-64. Then we need to prove important lemmas. How this was done is the main subject of this paper.

Floating point numbers can be stored either in floating point registers or in memory, and in each case we cannot always assume the encoding is irredundant (i.e. there may be several different encodings of the same real value, even apart from IEEE signed zeros). Thus, we need to take particular care over the distinction between values and their floating point encodings.[3] Systematically making this separation nicely divides our formalization into two parts: those that are concerned only with real numbers, and those where the floating point encodings with the associated plethora of special cases (infinities, NaNs, signed zeros etc.) come into play.

2 Floating Point Formats

Floating point numbers, at least in conventional binary formats, are those of the form $2^e k$ with the *exponent* e subject to a certain bound, and the *fraction* (also called significand or mantissa) k expressible in a binary positional representation using a certain number p of bits. The bound on the exponent range together with the allowed *precision* p determines a particular floating point *format*.

Floating point numbers cover a wide range of values from the very small to the very large. They are evenly spaced except that at the points 2^j the interval between adjacent numbers doubles. The intervals $2^j \quad x \quad 2^{j+1}$, possibly excluding one or both endpoints, are often called *binades*, and the numbers 2^j *binade boundaries*. In a decimal analogy, the gap between 1:00 and 1:01 is ten times the gap between 0:999 and 1:00, where all numbers are constrained to three significant digits. The following diagram illustrates this.



Our formalization of the encoding-free parts of the standard is highly generic, covering an infinite collection of possible floating point formats, even including absurd formats with zero precision (no fraction bits). It is a matter of taste whether the pathological cases should be excluded at the outset. We sometimes

[3] In the actual standard (p7) 'a bit-string is not always distinguished from a number it may represent'.

need to exclude them from particular theorems, but many of the theorems turn out to be degenerately true even for extreme values.

Section 3.1 of the standard parametrizes floating point formats by precision p and maximum and minimum exponents E_{max} and E_{min}. We follow this closely, except we represent the fraction by an integer rather than a value 1 $f < 2$, and the exponent range by two nonnegative numbers N and E. The allowable floating point numbers are then of the form $2^{e-N}k$ with $k < 2^p$ and 0 $e < E$. This was not done because of the use of biasing in actual floating point encodings (as we have stressed before, we avoid such issues at this stage), but rather to use nonnegative integers everywhere and carry around fewer side-conditions. The cost of this is that one needs to remember the bias when considering the exponents of floating point numbers. We name the fields of a triple as follows:

```
|- exprange (E,p,N) = E

|- precision (E,p,N) = p

|- ulpscale (E,p,N) = N
```

and the definition of the set of real numbers corresponding to a triple is:[4]

```
|- format (E,p,N) =
     f x | 9s e k. s < 2 ^ e < E ^ k < 2 EXP p ^
                   (x = --(&1) pow s * &2 pow e * &k / &2 pow N)g
```

This says exactly that the format is the set of real numbers representable in the form $(-1)^s 2^{e-N}k$ with $e < E$ and $k < 2^p$ (the additional restriction $s < 2$ is just a convenience). For many purposes, including floating point rounding, we also consider an analogous format with an exponent range unbounded above. This is defined by simply dropping the exponent restriction $e < E$. Note that the exponent is still bounded *below*, i.e. N is unchanged.

```
|- iformat (E,p,N) =
     f x | 9s e k. s < 2 ^ k < 2 EXP p ^
                   (x = --(&1) pow s * &2 pow e * &k / &2 pow N)g
```

We then prove various easy lemmas, e.g.

```
|- &0 IN iformat fmt

|- --x IN iformat fmt = x IN iformat fmt

|- x IN iformat fmt =) (&2 pow n * x) IN iformat fmt
```

[4] The ampersand denotes the injection from ℕ to ℝ, which HOL's type system distinguishes. The function EXP denotes exponentiation on naturals, and pow the analogous function on reals.

The above definitions consider the mere existence of triples (s,e,k) that yield the desired value. In general there can be many such triples that give the same value. However there is the possibility of a *canonical* representation:

```
|- iformat (E,p,N) =
      f x | 9s e k. (2 EXP (p - 1) <= k _ (e = 0)) ^
                    s < 2 ^ k < 2 EXP p ^
                    (x = --(&1) pow s * &2 pow e * &k / &2 pow N)g
```

This justifies our defining a ‘decoding’ of a representable real number into a standard choice of sign, exponent and fraction. This is defined using the Hilbert ″ operator and from the definition we derive:

```
|- x IN iformat(E,p,N)
   =) (2 EXP (p - 1) <= decode_fraction (E,p,N) x _
       (decode_exponent (E,p,N) x = 0)) ^
      decode_sign (E,p,N) x < 2 ^
      decode_fraction (E,p,N) x < 2 EXP p ^
      (x = --(&1) pow (decode_sign (E,p,N) x) *
           &2 pow (decode_exponent (E,p,N) x) *
           &(decode_fraction (E,p,N) x) / &2 pow N)
```

Note that it is these canonical notions, not the fields of any encodings, that we later discuss when we consider, say, whether the fraction of a number is even in rounding to nearest.

We prove that there can only be one restricted triple (s,e,k) for a given value, except for differently signed zeros, and these coincide with the canonical decodings defined above. For example:

```
|- s < 2 ^ k < 2 EXP p ^
   (2 EXP (p - 1) <= k _ (e = 0)) ^
   (x = --(&1) pow s * &2 pow e * &k / &2 pow N)
   =) (decode_fraction(E,p,N) x = k)
```

Nonzero numbers represented by a canonical triple such that $k < 2^{p-1}$ (and hence with $e = 0$) are often said to be *denormal* or *unnormal*. Other representable values are said to be *normal*. We do not define these terms formally in HOL at this stage, reserving them for properties of actual floating point register encodings, where a subtle terminological distinction is made between ‘denormal’ and ‘unnormal’ numbers. But we do now define criteria for an arbitrary real to be in the ‘normalized’ or ‘tiny’ range and these are used quite extensively later:

```
|- normalizes fmt x =
      (x = &0) _
      &2 pow (precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(x)

|- tiny fmt x = :(normalizes fmt x)
```

3 Units in the Last Place

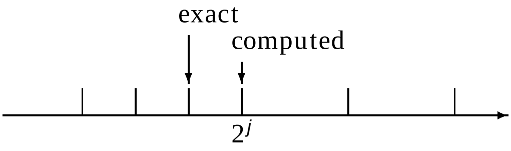
The term ‘unit in the last place’ is only mentioned in passing by the standard on p. 12 when discussing binary to decimal conversion. Nevertheless, it is of great importance for later proofs because the error bounds for transcendental functions need to be expressed in terms of ulps. Doing so is quite standard, yet there is widespread confusion about what an ulp is, and a variety of incompatible definitions appear in the literature.

Suppose $x \in \mathbb{R}$ is approximated by $a \in \mathbb{R}$, the latter being representable by a floating point number. For example, x might be the true result of a mathematical function, and a the approximation returned by a floating point operation. What do we mean by saying that the error $|x - a|$ is within $n\ ulp$? In the context of a finite binary (or decimal) string, a unit in the last place is naturally understood as the magnitude of its least significant digit, or in other words, the distance between the floating point number a and the next floating point number of greater magnitude. Indeed, if we examine two standard references on the subject, we see the definition framed in both ways:

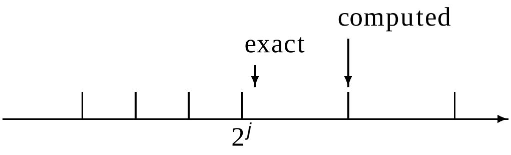
> In general, if the floating-point number $d.d \cdots d \times \beta^e$ is used to represent z, it is in error by $|d.d \cdots d - (z/\beta^e)|\beta^{p-1}$ units in the last place.[5] (Goldberg [5].)

> The term *ulp(x) (*for *unit in the last place*) denotes the distance between the two floating point numbers that are closest to x. (Müller [13].)

Both these definitions have some counterintuitive properties. For example, the following approximation is in error by $0.5ulp$ according to Goldberg, but intuitively, and according to Müller, $1ulp$:



But Müller’s definition has the somewhat curious property that its discontinuities occur away from the binade boundaries 2^j, because the closest floating point numbers to a real number x may not be the straddling ones. For example, the following (a valid rounding up) has an error of about $1.4ulp$ according to Müller, but intuitively and according to the Goldberg definition, it is less than 1.



[5] where p is the precision and β the base of the floating point format. For us $\beta = 2$.

Arguably no simple function either of the exact or computed result can avoid counterintuitive properties completely. However, it is very convenient to have such a simple definition. We adopt a definition more like Müller's in that it (a) is a function of the exact value, and (b) it includes a point 2^j in the interval immediately below it. However we effectively insist that an ulp in x is the distance between the two closest *straddling* floating point numbers a and b, i.e. those with $a \quad x \quad b$ and $a \neq b$ assuming an unbounded exponent range.

This seems to convey the natural intuition of units in the last place, and preserves the important mathematical properties that rounding to nearest corresponds to an error of $0{:}5ulp$ and directed roundings imply a maximum error of $1ulp$. The actual HOL definition is explicitly in terms of binades, and defined using the Hilbert choice operator ″:[6]

```
|- binade(E,p,N) x =
      ″e. abs(x) <= &2 pow (e + p) / &2 pow N ^
          8e′. abs(x) <= &2 pow (e′ + p) / &2 pow N =) e <= e′

|- ulp(E,p,N) x = &2 pow (binade(E,p,N) x) / &2 pow N
```

After a fairly tedious series of proofs, we eventually derive the theorem that an ulp does indeed yield the distance between two straddling floating point numbers:

```
|- :(p = 0)
   =) 9a b. a IN iformat(E,p,N) ^ b IN iformat(E,p,N) ^
            a <= x ^ x <= b ^ (b = a + ulp(E,p,N) x) ^
            :(9c. c IN iformat(E,p,N) ^ a < c ^ c < b)
```

4 Rounding

Floating point rounding takes an arbitrary real number and chooses a floating point approximation. Rounding is regarded in the Standard as an operation mapping a real to a member of the extended real line $\mathbb{R}$ [f+ 1 ; − 1 g, not the space of floating point numbers itself. Thus, encoding and representational issues (e.g. zero signs) are not relevant to rounding. The Standard defines four rounding modes, which we formalize as the members of an enumerated type:

```
roundmode = Nearest | Down | Up | Zero
```

Our formalization defines rounding into a given format as an operation that maps into the corresponding format *with an exponent range unbounded above*. That is, we do not take any special measures like coercing overflows back into the format or to additional 'infinite' elements; this is defined separately when we consider operations. While this separation is not quite faithful to the letter of the

[6] Read '″e. ...' as 'the e such that ...'.

Standard, we consider our approach preferable. It has obvious technical convenience, avoiding the formally laborious adjunction of infinite elements to the real line and messy side-conditions in some theorems about rounding. Moreover, it avoids duplication of closely related issues in different parts of the Standard. For example, the rather involved criterion for rounding to $\mathit{1}$ in round-to-nearest mode in sec. 4.1 of the Standard ('an infinitely precise result with magnitude at least $E_{max}(2 - 2^{-p})$ shall round to $\mathit{1}$ with no change of sign') is not needed. In our setup we later consider numbers that round to values outside the range-restricted format as overflowing, so the exact same condition is implied. This approach in any case *is* used later in the Standard 7.3 when discussing the raising of the overflow exception ('. . . were the exponent range unbounded').

Rounding is defined in HOL as a direct transcription of the Standard's definition. There is one clause for each of the four rounding modes:

```
|- (round fmt Nearest x =
      closest_such (iformat fmt) (EVEN o decode_fraction fmt) x) ^
   (round fmt Down x = closest a | a IN iformat fmt ^ a <= x x) ^
   (round fmt Up x = closest a | a IN iformat fmt ^ a >= x x) ^
   (round fmt Zero x =
      closest a | a IN iformat fmt ^ abs a <= abs x x)
```

For example, the result of rounding `x` down is defined to be the closest to `x` of the set of real numbers `a` representable in the format concerned (`a IN iformat fmt`) and no larger than `x` (`a <= x`). The subsidiary notion of 'the closest member of a set of real numbers' is defined using the Hilbert " operator. As can be seen from the definition, rounding to nearest uses a slightly elaborated notion of closeness where the result with an even fraction is preferred.[7]

```
|- is_closest s x a =
      a IN s ^ 8b. b IN s =) abs(b - x) >= abs(a - x)

|- closest s x = "a. is_closest s x a

|- closest_such s p x =
     "a. is_closest s x a ^ (8b. is_closest s x b ^ p b =) p a)
```

In order to derive useful consequences from the definition, we then need to show that the postulated closest elements always exist. Actually, this depends on the format being nontrivial. For example, if the format has nonzero precision, then rounding up behaves as expected:

[7] Note again the important distinction between real values and encodings. The canonical fraction is used; the question of whether the actual floating point value has an even fraction is irrelevant.

```
|- :(precision fmt = 0)
   =) round fmt Up x IN iformat fmt ^
       x <= round fmt Up x ^
       abs(x - round fmt Up x) < ulp fmt x ^
       8c. c IN iformat fmt ^ x <= c
           =) abs(x - round fmt Up x) <= abs(x - c)
```

The strongest results for rounding to nearest depend on the precision being at least 2. This is because in a format with $p = 1$ nonzero normalized numbers all have fraction 1, so ‘rounding to even’ no longer discriminates between adjacent floating point numbers in the same way.

4.1 Lemmas about Rounding

While these results are the key to all properties of rounding, there are lots of other important consequences of the definitions that we sometimes use in proofs. For example, rounding is monotonic in all modes:

```
|- :(precision fmt = 0) ^ x <= y =) round fmt rc x <= round fmt rc y
```

and has various properties like the following:

```
|- :(precision fmt = 0) ^ a IN iformat fmt ^ a <= x
   =) a <= round fmt rc x

|- :(precision fmt = 0) ^ a IN iformat fmt ^ abs(x) <= abs(a)
   =) abs(round fmt rc x) <= abs(a)
```

Something already representable rounds to itself, and conversely:

```
|- a IN iformat fmt =) (round fmt rc a = a)

|- :(precision fmt = 0)
   =) ((round fmt rc x = x) = x IN iformat fmt)
```

An important case where a result of a calculation *is* representable is subtraction of nearby quantities.

```
|- a IN iformat fmt ^ b IN iformat fmt ^ a / &2 <= b ^ b <= &2 * a
   =) (b - a) IN iformat fmt
```

This well-known result [5] can be generalized to subtraction of nearby quantities in formats with more precision (as effectively occur in the intermediate step of an `fma` operation):

```
|- :(p = 0) ^
   a IN iformat (E1,p+k,N) ^
   b IN iformat (E1,p+k,N) ^
   abs(b - a) <= abs(b) / &2 pow (k + 1)
   =) (b - a) IN iformat (E2,p,N)
```

A little thought shows that the first version is easily derivable by linear arithmetic reasoning (automatic in HOL) from this more general version with k = 1. Both the general and special case can be strengthened if both inputs are known to be in the same binade, e.g.

```
|- :(p = 0) ^
   a IN iformat (E1,p+k,N) ^ b IN iformat (E1,p+k,N) ^
   abs(b - a) <= abs(b) / &2 pow k ^
   (8e. &2 pow e / &2 pow N <= abs(b) = &2 pow e / &2 pow N <= abs(a))
   =) (b - a) IN iformat (E2,p,N)
```

We have also proved some direct cancellation theorems for an `fma` operation. The following embodies the useful fact that one can get an exact representation of a product in two parts by one multiplication and a subsequent `fma` to get a correction term.[8]

```
|- a IN iformat fmt ^ b IN iformat fmt ^
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(a * b)
   =) (a * b - round fmt Nearest (a * b)) IN iformat fmt
```

A few other miscellaneous theorems about rounding include simple relations between rounding modes, e.g.

```
|- :(precision fmt = 0) =) (round fmt Down (--x) = --(round fmt Up x))
```

Plenty of other lemmas are proved formally too.

4.2 Rounding Error

One of the central questions in floating point error analysis is the bounding of rounding error. We define rounding error as:

```
|- error fmt rc x = round fmt rc x - x
```

The rounding error is easily bounded in terms of ulps:

```
|- :(precision fmt = 0)
   =) (abs(error fmt Nearest x) <= ulp fmt x / &2) ^
      (abs(error fmt Down x) < ulp fmt x) ^
      (abs(error fmt Up x) < ulp fmt x) ^
      (abs(error fmt Zero x) < ulp fmt x)
```

[8] See `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps` for example; thanks to Paul Miner for pointing us at these documents.

and ulps in their turn can be bounded in terms of relative error, provided denormalization is avoided:

```
|- normalizes fmt x ^ :(precision fmt = 0) ^ :(x = &0)
   =) ulp fmt x <= abs(x) / &2 pow (precision fmt - 1)
```

Conversely, we have a lower bound on ulps in terms of relative error:

```
|- abs(x) / &2 pow (precision fmt) <= ulp fmt x
```

A simple generic result for all rounding modes can be stated in terms of a parameter mu:

```
|- (mu Nearest = &1 / &2) ^
   (mu Down = &1) ^
   (mu Up = &1) ^
   (mu Zero = &1)
```

namely:

```
|- normalizes fmt x ^ :(precision fmt = 0)
   =) abs(error fmt rc x)
        <= mu rc * abs(x) / &2 pow (precision fmt - 1)
```

5 Exceptions and Flag Settings

The IEEE operations not only return values, but also indicate special conditions by setting sticky flags or raising exceptions. These indications are all triggered according to fairly straightforward criteria. For example, the overflow flag is set precisely when the result rounded (as we do anyway) with unbounded upper exponent range is not a member of the actual format with bounded exponent range. Similarly, the inexact flag is set either if overflow occurs or if rounding was nontrivial, i.e. the rounded number was not already representable. It is easy to state these two criteria in terms of existing concepts, e.g.

```
let overflow_flag = :(round fmt rc x IN format fmt) in
let inexact_flag = overflow _ :(round fmt rc x = x) in
...
```

5.1 Underflow

Somewhat more complicated is the definition of underflow.[9] The Standard (sec. 7.4) underspecifies underflow considerably, so it is possible that different implementations of the Standard could set flags or raise exceptions differently. Underflow is said to occur when there is both *tininess* and *loss of accuracy*, and each of these may be detected in two different ways. We have already defined tininess of a number, but the number tested for tininess may either be:

[9] An additional complication is that the criteria for flag-setting and exception-raising are different. We consider only flag setting here.

{ The exact result before any rounding.

{ The result rounded as if the exponent range were unbounded *below*.

While we already round into a format with the exponent range unbounded *above*, we have no easy way of using our existing infrastructure to define rounding with the exponent range unbounded *below*. Instead, we consider rounding with 'sufficiently large lower exponent range':

```
|- tiny_after_rounding fmt rc x =
      9N. N > ulpscale fmt ^
          tiny fmt (round(exprange fmt,precision fmt,N) rc x)
```

Since this is not a direct transcription of the Standard, we have proved a number of 'sanity check' lemmas to make it clear that this definition is equivalent to the Standard's definition. The crucial one is that if a result is tiny when rounded with a particular lower exponent range, then it will still be tiny for all larger lower exponent ranges:

```
|- 2 <= precision fmt
    =) (tiny_after_rounding fmt rc x =
          9N. N > ulpscale fmt ^
              8M. M >= N
                  =) tiny fmt
                        (round(exprange fmt,precision fmt,M) rc x))
```

Loss of accuracy may also be detected in more than one way, either as simple inexactness (see above) or as a difference between the actually rounded result and the result rounded as if the exponent range were unbounded below. Again we state a 'profinite' version of this definition and again feel honor-bound to justify it by some additional lemmas.

```
|- losing fmt rc x =
      9N. N > ulpscale fmt ^
          :(round (exprange fmt,precision fmt,N) rc x = round fmt rc x)
```

5.2 Relations between Underflow Conceptions

Using the definitions of the previous section, it is easy to define underflow in any of the ways the Standard allows, including the choice adopted in IA-64. It is of interest to note, however, that there are very strong correlations between the different criteria for tininess and loss of accuracy. In fact, one of them implies all the others (for reasonable formats), as we have formally proved in HOL:

```
|- 2 <= precision fmt ^ losing fmt rc x
   =) tiny_after_rounding fmt rc x

|- :(precision fmt = 0) ^ tiny_after_rounding fmt rc x
   =) tiny fmt x

|- :(precision fmt = 0) ^ losing fmt rc x
   =) :(round fmt rc x = x)
```

Thus, while an implementation may make a variety of choices, many of the combinations collapse into the same one when their meaning is considered. Since the above theorems show that `losing` is the ‘weakest’ criterion for underflow, it is occasionally worth strengthening some previous theorems to take it as a hypothesis, e.g:

```
|- :(losing fmt rc x) ^ :(precision fmt = 0)
   =) abs(error fmt rc x)
       <= mu rc * abs(x) / &2 pow (precision fmt - 1)
```

The following very useful theorem can be employed to show that the rounding of an `fma` operation does not underflow provided the argument being added is sufficiently far from the low end:

```
|- :(precision fmt = 0) ^
   a IN iformat fmt ^ b IN iformat fmt ^ c IN iformat fmt ^
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(c)
   =) :(losing fmt rc (a * b + c))
```

5.3 Flag Settings for Perfect Rounding

Certain software algorithms, e.g. those for division suggested in [11], are designed so that they set no flags and trigger no exceptions in intermediate stages, and culminate in a single `fma` operation that is supposed to deliver the correctly rounded result and set most of the flags, including overflow, underflow and inexact. It is a useful observation in proofs that it suffices to verify only that the result is correct, in the precise sense that the ideal value and the value computed before the final rounding will round identically in all rounding modes. The correctness of these three flags then follows immediately, as does the correct sign of zero results. Properly speaking, in the case of underflow, one needs to prove perfect rounding even assuming unbounded exponent range, but this is usually a straightforward extension. The only case that requires some thought is inexactness, which comes down to the following theorem:

```
|- :(precision fmt = 0) ^
   (8rc. round fmt rc x = round fmt rc y)
   =) 8rc. (round fmt rc x = x) = (round fmt rc y = y)
```

Note that the theorem is symmetrical between `x` and `y`, so it suffices to prove that, in any given rounding mode, if `x` rounds to itself, so does `y`. The proof is simple: if `x` rounds to itself, then it must be representable. But by hypothesis, `y` rounds to the same thing, that is `x`, in *all rounding modes*. In particular the roundings up and down imply `x <= y` and `x >= y`, so `y = x`.

Overflow is detected after rounding, so it is immediate that if `x` and `y` round identically, they will either both overflow or both not overflow. Similarly, it is easy to see that underflow behavior is equivalent.[10] For the signs of zeros, it suffices to prove:

```
|- :(precision fmt = 0) ^
   (8rc. round fmt rc x = round fmt rc y)
   =) (x > &0 = y > &0) ^ (x < &0 = y < &0)
```

This follows easily from that fact that zero is always representable in any format. For example, if `x` is (strictly) positive, it must round to a strictly positive number in round-up mode. Thus so must `y`, so `y` must also be strictly positive. The other cases are analogous.

6 Encodings

IA-64 includes direct support for several different floating point formats including an internal 82-bit format with a 17-bit exponent and 64-bit fraction (with an explicit 1 bit). Our formalization uses a single HOL type `float`, and all the available floating point numbers can be mapped into this type.

The Standard includes a variety of special numbers such as infinities and NaNs. The subset of 'sensible' values is defined in the HOL formalization by a predicate `finite:float->bool`. It is mainly with numbers in this subset that we will be concerned. The real value of a floating point number is defined by a HOL function `Val:float->real`. Again, we will not show the definition here.

7 Operations

The operations such as addition, subtraction and multiplication are defined in the Standard by composing previously defined concepts in a straightforward way. Roughly speaking, special inputs are treated in some reasonable way, while for finite inputs, the result is generated as if the exact answer were calculated and rounded, with appropriate flag settings. Certain value coercions also happen on overflow, depending on the rounding mode. We will not show the precise definition of the IA-64 operations, but only indicate some respects in which the definitions require care.

[10] Actually we have only proved this for the IA-64 definition of underflow, but other variants would work too.

The IEEE standard does not explicitly address the `fma` operation. Generally speaking, one can extrapolate straightforwardly from the IEEE-754 specifications of addition and multiplication operations. There are some debatable questions, however, mostly connected with the signs of zeros. First, the interpretation of addition and multiplication as degenerate cases of `fma` requires some policy on the sign of $1 \times -0 + 0$. More significantly, the `fma` leads to a new possibility: $a \times b + c$ can round to zero even though the exact result is nonzero. Out of the operations in the standard, this can occur for multiplication or division, but in this case the rules for signs are simple and natural. A little reflection shows that this cannot happen for pure addition, so the rule in the standard that 'the sign of a sum ... differs from at most one of the addend's signs' is enough to fix the sign of zeros when the exact result is nonzero. For the `fma` this is not the case, and IA-64 guarantees that the sign correctly reflects the sign of the exact result in such cases. This is important, for example, in ensuring that certain software algorithms yield the correctly signed zeros in all cases without special measures.

8 Proof Tools

The formal theorems proved above capture the main general lemmas about floating point arithmetic that have been found important in the verification undertakings to date. However, it is often important to have special theorem-proving tools based around (variants of) the lemmas, to avoid the tedium of manually applying them and proving routine side-conditions. Broadly speaking, the operations that are most important to automate involve 'symbolic execution' of various kinds.

8.1 Explicit Execution

It often happens that one needs to 'evaluate' floating point operations or associated formal concepts for particular explicit values. For example, one often wants to:

- Calculate `ulp(r)` for a particular rational number `r`.
- Calculate `round fmt rc r` for a particular floating point format `fmt`, rounding mode `rc` and rational number `r`.
- Evaluate `Val(a)` for a particular floating point number `a`.
- Prove that a particular floating point value is non-exceptional, i.e. return a theorem `|- finite(a)` for a particular floating point number `a`.

We have implemented HOL *conversions* (see [15] for more on conversions) to do all these, and a few other operations too. Now, explicit details of this sort can be disposed of automatically. For example, the conversion `ROUND_CONV` takes rounding parameters and a rational number to be rounded and not only returns the 'answer', but also a formally proved theorem that the answer is correct. (Under the surface, theorems about the uniqueness of rounding are applied.)

```
#ROUND_CONV ‘round (10,11,12) Nearest (&22 / &7)‘;;
it : thm = |- round (10,11,12) Nearest (&22 / &7) = &1609 / &512
```

HOL already includes proof tools to perform explicit calculation with rational numbers and even with computable real numbers [8]. In conjunction with the new proof tools, we now have powerful automatic assistance for goals involving all forms of explicit calculation.

8.2 Automated Error Analysis

Explicit computations are not always enough. Sometimes one does not know the actual floating point values involved, merely some properties such as maximum or minimum magnitudes and the maximum absolute or relative error from some ‘ideal’ value. We have implemented HOL tools to propagate knowledge of such properties through additional `fma` operations. Using these, it is simple to get a formally proven absolute or relative error bound for a sequence of `fma` operations, e.g. the evaluation of a polynomial approximation to a transcendental function, completely automatically, given only some assumptions on the input number(s).

Whether one wants absolute or relative error depends on the kind of proofs being undertaken. When trying to get a sharp ulp error bound for a transcendental function approximation, we find it useful to split the ideal output into binades (determining the ulp value), and evaluate the maximum absolute error on each corresponding input set. We can then see which binade yields the largest ulp error, without the loosening of the error bound that would be caused by using relative error. However, typically the errors are large only for a few binades, so we still use relative error to dispose of all the others, for efficiency reasons. (For extended precision, there could be around 2^{15} different binades.)

The proof tool for absolute errors requires theorems about each nonconstant input `x` to an `fma` operation of the following form: [11]

```
|- finite x

|- abs(Val x) <= b

|- abs(Val x - y) <= e
```

Here `b` and `e` must be expressions made up of rational constants, while `y`, the value approximated, can be any expression. From this information, the proof tool automatically derives analogous assertions for the output of the `fma` operation. At present, fairly crude maximization techniques are used to evaluate the range and error in the output. This has proved fine for verifications undertaken to date, since the intermediate inputs tend to be monotonic over the fairly narrow intervals considered. However, we are presently considering a more sophisticated mechanism to get tighter error bounds.

[11] Assumptions of this sort are only needed for variables, as they are derived automatically for explicit values as described in the previous section.

For relative error, the approach is analogous, with the absolute error e replaced by a relative error. Moreover, an additional assumption is needed about the *minimum* (nonzero) size of the input, because to get a sharp relative error result we need to prove that underflow doesn't occur. We use the following definition:

```
|- zorbigger a x = &0 <= a ^ ((x = &0) _ a <= abs(x))
```

It is straightforward to propagate such assumptions through expressions for varying threshold a, using theorems such as the following:

```
|- zorbigger a1 x1 ^ zorbigger a2 x2 =) zorbigger (a1 * a2) (x1 * x2)

|- x1 IN iformat fmt ^ x2 IN iformat fmt ^ x3 IN iformat fmt ^
   zorbigger a3 x3 ^ :(x3 = &0)
   =) zorbigger (a3 / &2 pow (2 * precision fmt)) (x1 * x2 + x3)
```

and then when required we can derive normalization from the lemma:

```
|- normalizes fmt =
   zorbigger (&2 pow (precision fmt - 1) / &2 pow ulpscale fmt)
```

8.3 Intermediate Levels of Explicitness

There are some other possibilities that fall between the previous categories. For example, we have formally checked some correctness proofs for floating point square root algorithms using a methodology discussed in [2]. This methodology gives us a proof of correctness for all but a certain set of values, isolated using number-theoretic considerations. It is then necessary, to get an overall correctness proof, to check the remaining values explicitly.

While in principle this can be done with explicit calculation, such an approach is inefficient and unnatural, because the set of values is parametrized by a much smaller set of e_i and k_i by simply varying the exponent while preserving its even/odd parity:

$$2^{e_i+2n}k_i$$

It is much more natural to check the values only for a particular n, say $n = 0$, and then extrapolate from that. This can be justified by scaling theorems for rounding, provided overflow cannot occur for the maximum exponent. The scaling theorem for rounding also requires that loss of precision is avoided; one sufficient condition is shown in the next theorem.

```
|- 2 <= precision fmt ^
   (&2 pow (precision fmt - 1) / &2 pow ulpscale fmt <= abs x _
    (round fmt rc x = x))
   =) (round fmt rc (&2 pow n * x) = &2 pow n * round fmt rc x)
```

At present, we have not automated this kind of scaling analysis completely, but it has been taken far enough that the proofs were all reasonably straightforward. If we did many more proofs of the same kind, further effort would be needed.

9 Conclusions and Related Work

We have detailed a theory of floating point arithmetic that is generic over a wide variety of floating point formats, and has then been specialized to the particular formats used in IA-64. By contrast, an earlier formalization by ourselves [7] required duplication of results for different precisions and did not achieve the same neat separation between floating point values and their encodings. Our present formulation contains a far larger collection of medium-level lemmas than any other formalization we are aware of. In contrast to some previous IEEE-754 specifications such as one in Z [1], ours is completely formal and all results have been logically proved by machine.

Most of the definitions (excluding, perhaps, some of those connected with underflow) are a direct formal translation of the Standard, making their correctness highly intuitive. For example, our definition of floating point rounding is the same as the Standard's, whereas all related machine-checked formalizations of which we are aware [12,14,16] use less intuitive translations. In some cases, this is forced by the limited mathematical expressiveness of other theorem provers.

The price one pays for intuitive high-level specifications is that one cannot automatically 'execute' the formal specification in the proof process. By contrast, ACL2 specifications like Russinoff's [16] are always executable by construction. We have ameliorated this shortcoming by providing a suite of automatic proof tools that can, effectively, execute specifications, and moreover can do more sophisticated forms of symbolic evaluation including automatic error analysis of chains of floating point computations. Since such 'execution' merely abbreviates and automates standard logical inferences, we have the advantage of generating a formal proof rather than relying on a separate execution mechanism. The only drawback of this is that using standard logical inferences is relatively slow.

The formalization described here has been used quite extensively in verification of various algorithms for division and square root [3] and some transcendental functions [17]. It is hoped that we can describe these verifications in more detail at a later date. Such verifications sometimes combine nontrivial continuous mathematics with low-level machine details and a certain amount of explicit execution. Using a general theorem prover like HOL equipped with our formalization and proof tools, all these disparate aspects can be unified in one system, and the final result verified according to the strictest standards of logical rigor. For example, one of the transcendental function verifications involves approximately 77 million primitive logical inferences. However, generating such big proofs is quite feasible as most of the more tedious parts are automatic.

References

1. M. Barratt. Formal methods applied to a floating-point system. *IEEE Transactions on Software Engineering*, 15:611{621, 1989.
2. M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1{11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.
3. M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In Koren and Kornerup [10], pages 96{105.
4. C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 64(7):24{32, July 1998.
5. D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23:5{48, 1991.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. J. Harrison. Floating point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.
8. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
9. IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.
10. I. Koren and P. Kornerup, editors. *Proceedings, 14th IEEE symposium on on computer arithmetic*, Adelaide, Australia, 1999. IEEE Computer Society.
11. P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111{119, 1990.
12. P. S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical memorandum 110167, NASA Langley Research Center, Hampton, VA 23681-0001, USA, 1995.
13. J-M. Muller. *Elementary functions: Algorithms and Implementation*. Birkhäuser, 1997.
14. J. O'Leary, X. Zhao, R. Gerth, and C-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1{14, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.
15. L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119{149, 1983.
16. D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148{200, 1998. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
17. S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In Koren and Kornerup [10], pages 4{11.

Universal Algebra in Type Theory

Venanzio Capretta

Computer Science Institute
University of Nijmegen
venanzio@cs.kun.nl

Abstract. We present a development of Universal Algebra inside Type Theory, formalized using the proof assistant Coq. We define the notion of a signature and of an algebra over a signature. We use setoids, i.e. types endowed with an arbitrary equivalence relation, as carriers for algebras. In this way it is possible to define the quotient of an algebra by a congruence. Standard constructions over algebras are defined and their basic properties are proved formally. To overcome the problem of defining term algebras in a uniform way, we use types of trees that generalize wellorderings. Our implementation gives tools to define new algebraic structures, to manipulate them and to prove their properties.

1 Introduction

The development of mathematical theories inside Type Theory presents some technical problems that make it difficult to translate an informal mathematical proof into a formalized one. In trying to carry out such a translation, one soon realizes that notions that were considered non-problematic and obvious at the informal level need a delicate formal analysis. Additional work is often needed just to define the mathematical structures under study and the basic tools to manipulate them. Besides the difficulty of rendering exactly what is expressed only in intuitive terms, there is the non-trivial task of translating into Type Theory what was originally intended to be expressed inside some form of set theory (for example in ZF). This paper presents a development of such tools for generic algebraic reasoning, which has been completely formalized in the Coq proof development system (see [3]). We want to enable the users of such tools to easily define their own algebraic structures, manipulate objects and reason about them in a way that is not too far from ordinary mathematical practice.

Our work stemmed from an original project of formal verification of Computer Algebra algorithms in Type Theory. We realized then that the definition of common mathematical structures, like those of ring and field, together with tools to manipulate them, was essential to the success of the enterprise. We decided to develop Universal Algebra as a general tool to define algebraic structures.

Previous work on Algebra in Type Theory was done by Paul Jackson using the proof system Nuprl (see [14]), by Peter Aczel on Galois Theory (see [1]) and by Huet and Saïbi on Category Theory (see [13]). A large class of algebraic structures has been developed in Coq by Loïc Pottier.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 131–148, 1999.

Another aim is the use of a two level approach to the derivation of propositions about algebraic objects (see [4]). In this approach, statements about objects are lifted to a syntactic level where they can be manipulated by operators. An example is the simplification of expressions and automatic equational reasoning. This method was already used by Douglas Howe to construct a partial syntactic model of the Type Theory of Nuprl inside Nuprl itself, which can be used to program tactics inside the system (see [12]). An application of this reflection mechanism to algebra was developed by Samuel Boutin in Coq for the simplification of ring expressions (see [5]). In the present work the need to parameterize the construction of the syntactic level on the type of signatures posed an additional problem. A very general type construction similar to Martin-Löf's Wellorderings was employed for the purpose.

Finally, the study of the computational content of algebras is particularly interesting. We investigate to what extent algebraic objects can be automatically manipulated inside a proof checker. This can be done through the use of certified versions of algorithms borrowed from Computer Algebra, as was done by Thery in [23] and by Coquand and Persson in [8] for Buchberger's algorithm.

The files of the implementation are available via the Internet at the site *http://www.cs.kun.nl/~venanzio/universal_algebra.html.*

Type Theory and Coq. The work presented here has been completely formalized inside Coq, but it could have equally easily been formalized in other proof systems based on Type Theory, like Lego or Alf. Although Coq is based on the Extended Calculus of Constructions (see [15]) everything could be formalized in a weaker system. Any Pure Type System that is at least as expressive as PT (see [2]) endowed with inductive types (see [22]), or Martin-Löf's Type Theory with at least two universes (see [16], [17] or [19]) is enough.

We assume that we have two universes of types ^{s} for sets and ^{p} for propositions (Set and Prop in the syntax of Coq), and that they both belong to the higher universe □ (Type in Coq). The product type $x : A:B$ is written using Coq notation $(x : A)B$. If $B :\ ^p$ we also write $(8x : A)B$. In Coq it is possible to define record types in which every field can depend on the values of the preceding fields. We will use the following notation for records.

$$Record\ Name : Type := constructor \begin{cases} field_1 : A_1 \\ \vdots \\ field_n : A_n \end{cases}$$

An element of this record type is in the form $(constructor\ a_1 ::: a_n)$ where $a_1 : A_1; a_2 : A_2[field_1 := a_1]; :::; a_n : A_n[field_1 := a_1; :::; field_{n-1} := a_{n-1}]$. We have the projections

$$\begin{array}{l} field_1 : Name\ !\ A_1 \\ field_2 : (x : Name)(A_2[field_1 := (field_1\ x)]) \\ \vdots \\ field_n : (x : Name)(A_n[field_1 := (field_1\ x)] ::: [field_{n-1} := (field_{n-1}\ x)]) \end{array}$$

In Coq a record type is a shorthand notation for an inductive type with only one constructor. In a system without this facility they could be represented by nested -types.

Algebraic structures in Type Theory. Let us start by considering a simple algebraic structure and its implementation in Type Theory. The standard mathematical definition of a group is the following.

Definition 1. *A* group *is a quadruple* $hG; \ ;e;\_^{-1}i$*, where* G *is a set,* $\ $ *a binary operation on* G*,* e *an element of* G *and* $\_^{-1}$ *a unary operation on* G *such that* $(x \ y) \ z = x \ (y \ z)$*,* $x \ e = x$ *and* $x \ (x^{-1}) = e$ *for all* $x;y;z \ 2 \ G$*.*

An immediate translation in Type Theory would employ a record type.

$$
\begin{array}{l}
Record\ Group : \square := \\
\quad group \left\{ \begin{array}{ll}
elements & : Setoid \\
operation & : elements\ !\ elements\ !\ elements \\
unit & : elements \\
inverse & : elements\ !\ elements
\end{array} \right.
\end{array}
$$

where a setoid is a set endowed with an equivalence relation (see next section).

But this is not yet enough since we didn't specify that the group axioms must be satisfied. This is usually done by enlarging the record to contain proofs of the axioms.

$$
\begin{array}{l}
Record\ Group : \square := \\
\quad group \left\{ \begin{array}{ll}
elements & : Setoid \\
operation & : elements\ !\ elements\ !\ elements \\
unit & : elements \\
inverse & : elements\ !\ elements \\
associativity & : (8x;y;z : elements) \\
 & \quad (operation\ (operation\ x\ y)\ z) \\
 & \quad = (operation\ x\ (operation\ y\ z)) \\
unitax & : (8x : elements)(operation\ x\ unit) = x \\
inverseax & : (8x : elements)(operation\ x\ (inverse\ x)) = unit
\end{array} \right.
\end{array}
$$

So to declare a specific group, for example the group of integers with the sum operation, we must specify all the fields:

$$Integer : Group := (group\ Z\ plus\ 0\ -\ p1\ p2\ p3):$$

where $p1$, $p2$, $p3$ are proofs of the axioms.

Why it is useful to develop Universal Algebra. Once an algebraic structure has been specified in this way, we proceed to give standard definitions like those of subgroup, product of groups, quotient of a group by a congruence relation, homomorphism of groups and we prove standard results. In this way many algebraic structures can be specified, and theorems can be proved about them (see the work by Lo¨c Pottier).

Since most of the definitions and basic properties are the same for every algebraic structure, having an abstract general formulation of them would save us from duplicating the same work many times. This is the main reason why it is interesting to develop Universal Algebra. To this aim we should internalize the generalization of the previous construction to have a general notion of algebraic structure inside Type Theory.

2 Setoids

Why we need setoids, informal definition of setoid. The first step before the implementation of Universal Algebra in Type Theory is to have a flexible translation of the intuitive notion of set. Interpreting sets as types would rise some problems: the structure of types is rather rigid and does not allow the formation of subtypes or quotient types. Since we need to define subalgebras and quotient algebras we are led to consider a more suitable solution. In some version of (extensional) type theory notions of subtype and quotient type are implemented (for example in the Nuprl system, see [6]), but the version of (intensional) type theory implemented in Coq does not. Nevertheless a model of extensional type theory inside intensional type theory has been constructed by Martin Hofmann (see [10]). We use a variant of this model, which has already been implemented by Huet and Saïbi in [13] and used by Pottier.

The elements of a type are build up using some constructors, and elements of a type are said to be equal when they are convertible. Thus a type cannot be defined by a predicate over an other type (subtyping) or by redefining the equality (quotienting). We allow ourselves to be more liberal with equality by defining a setoid to be a pair formed by a set and an equivalence relation over it. Thus we can quotient a setoid by just changing the equivalence relation. Subsetoids are obtained by quotienting Σ-types, i.e. if $S = \langle A, =_S \rangle$ is a setoid and P is a predicate over A (that is closed under $=_S$), we can define the subsetoid determined by P to be $S^P = \langle (\Sigma x : A.(P\ x)), =_{S^P} \rangle$ where $\langle a_1, p_1 \rangle =_{S^P} \langle a_2, p_2 \rangle$ iff $a_1 =_S a_2$.

Since we explicitly work with equivalence relations all the definitions on setoids (predicates over setoids, relations between setoids, setoids functions) must be required to be invariant under the given equality.

Formal definition of setoid.

Definition 2.

$$\mathit{Record\ Setoid} : \Box := \mathit{setoid} \begin{cases} \mathit{s\_el} & : \mathit{Set} \\ \mathit{s\_eq} & : \mathit{s\_el} \to \mathit{s\_el} \to \mathit{Prop} \\ \mathit{s\_proof} & : (\mathit{Equiv}\ \mathit{s\_eq}) \end{cases}$$

where $(\mathit{Equiv}\ \mathit{s\_eq})$ is the proposition stating that $\mathit{s\_eq}$ is an equivalence relation over the set $\mathit{s\_el}$.

We often identify a setoid S with its carrier set $(\mathit{s\_el}\ S)$. In Coq this identification is realized through the use of *implicit coercions* (see [21]). Similar implicit coercions are also used to identify an algebraic structure with its carrier. If $a, b : S$ (i.e. as we said $x, y : (\mathit{s\_el}\ S)$), we use the simple notation $x = y$ in place of

$(s\_eq\ x\ y)$; in Coq an infix operator [=] is defined so we can write x [=] y. As a general methodology if op is a set operator, we use the notation [op] for the corresponding setoid operator. Whenever we want to stress the setoid in which the equality holds (two setoids may have the same elements but different equalities) we write $x =_S y$.

Properties and constructions on setoids. As we have mentioned above, we have to be careful when dealing with constructions on setoids. For example, predicates, relations and functions should be invariant under the given equality.

Definition 3. *A predicate P over the carrier of a setoid S, i.e. $P : (s\_el\ S) \to \star^p$ is said to be* well defined *(with respect to $=_S$) if $(\forall x,y : S)x =_S y \to (P\ x) \to (P\ y).$ The type of* setoid predicates *over S is the record type*

$$\begin{array}{l} Record\ Setoid\_predicate\ : \square\ := \\ \quad setoid\_predicate \left\{ \begin{array}{l} sp\_pred\ \ : S \to \star^p \\ sp\_proof : (Predicate\_well\_defined\ sp\_pred) \end{array} \right. \end{array}$$

where $(Predicate\_well\_defined\ sp\_pred)$ is the above property.

Definition 4. *A relation R on the carrier of a setoid S, i.e. $P : (s\_el\ S) \to (s\_el\ S) \to \star^p$ is said to be* well defined *(with respect to $=_S$) if*

$$(\forall x_1,x_2,y_1,y_2 : S)x_1 =_S x_2 \to y_1 =_S y_2 \to (R\ x_1\ y_1) \to (R\ x_2\ y_2).$$

The type of setoid relations *on S is the record type*

$$\begin{array}{l} Record\ Setoid\_relation\ : \square\ := \\ \quad setoid\_relation \left\{ \begin{array}{l} sr\_rel\ \ \ : S \to S \to \star^p \\ sr\_proof : (Relation\_well\_defined\ sr\_rel) \end{array} \right. . \end{array}$$

By declaring two implicit coercions we can use setoid predicates and relations as if they were regular predicates and relations, i.e. if $P : (Setoid\_predicate\ S)$ and $x : S$ then $(P\ x)$ is a shorthand notation for $((sp\_pred\ P)\ x) : \star^p$ and if $R : (Setoid\_relation\ S)$ and $x,y : S$ then $(R\ x\ y)$ is a shorthand notation for $((sr\_rel\ R)\ x\ y) : \star^p$.

As we have mentioned in the informal discussion subsetoids can be defined by a setoid predicate by giving a suitable equivalence relation over a Σ-type.

Definition 5. *Let S be a setoid and P a setoid predicate over it. Then* the subsetoid of S separated by P *is the setoid $S|P$ that has carrier $\{x : S.(P\ x)\}$ and equality relation $\langle a_1,p_1\rangle =_{S|P} \langle a_1,p_1\rangle \iff a_1 =_S a_2$.*

Even easier is the definition of a quotient of a setoid by an equivalence (setoid) relation. It is enough to substitute such relation in place of the original equality.

Definition 6. *Let S be a setoid and $Eq : (Setoid\_relation\ S)$ such that $(sr\_rel\ Eq)$ is an equivalence relation on $(s\_el\ S)$. Then the* quotient setoid *S/Eq is the setoid with carrier set $(s\_el\ S)$ and equality relation Eq.*

Notice that the notion of quotient setoid is different from the notion of quotient set in set theory: the elements of S/Eq are not equivalence classes, as in set theory, but they are exactly the same as the elements of S.

Definition 7. *Let S_1 and S_2 be two setoids. Their* product *is the setoid $S_1[\times]S_2$ with carrier set $(s\_el\ S_1) \times (s\_el\ S_2)$ and equality relation*

$$\langle x_1, x_2\rangle =_{S_1[\times]S_2} \langle y_1, y_2\rangle \iff x_1 =_{S_1} y_1 \wedge x_2 =_{S_1} y_2.$$

Definition 8. *Let S_1 and S_2 be two setoids. The setoid of functions from S_1 to S_2 is the setoid $S_1[\rightarrow]S_2$ with carrier set the type of those functions between the two carriers that are well-defined with respect to the setoid equalities*

$$Record\ S_1[\rightarrow]S_2 := setoid\_function \left\{ \begin{array}{l} s\_function : S_1 \rightarrow S_2 \\ s\_fun\_proof : \\ \quad (fun\_well\_defined\ s\_function) \end{array} \right.$$

where $(fun\_well\_defined\ s\_function)$ is the proposition $(\forall x_1, x_2 : S_1) x_1 =_{S_1} x_2 \rightarrow (s\_function\ x_1) =_{S_2} (s\_function x_2)$, and $f =_{S_1[\rightarrow]S_2} g$ is the extensional equality relation $(\forall x : S_1)(f\ x) =_{S_2} (g\ x)$.

In a similar way we can define other constructions on setoids and define operators on them (see the source files for a complete list).

3 Signatures and Algebras

Using the development of setoids from the previous section as our notion of sets we can now translate Universal Algebra into Type Theory. We use as a guide the chapter on Universal Algebra by K. Meinke and J. V. Tucker from the *Handbook of Logic in Computer Science* ([18]). We differ from that work only in that we consider just finite signatures (so that they can be implemented by lists) and we do not require that carrier sets are non-empty. This second divergence is justified by the difference between first order predicate logic (which is the logic usually employed to reason about algebraic structures), that always assumes the universe of discourse to be non-empty, and Type Theory, in which this assumption is not present and, therefore, we can reason about empty structures (about this see also [2], section 5.4).

Definition of signature. We begin by defining the notion of a (many-sorted) signature. A signature is an abstract specification of the carrier sets (called *sorts*) and operations of an algebra, and it is given by the number of sorts n and a list of operation symbols $[f_1, \ldots, f_m]$ where each of the functions f_i must be specified by giving its type, i.e. by saying how many arguments the function has, to which one of the sorts each argument belongs and to which sort the result of the application of the operation belongs. Each sort is identified by an element of the finite set $\mathbb{N}_n = \{0, \ldots, n-1\}$ (in our Coq implementation $\mathbb{N}_n$ is represented by `(Finite n)` and its elements are represented by `n}-(0)`, `n}-(1)`, ..., `n}-(n-1)`).

As an example suppose we want to define a structure $hnat; bool; O; S; true; false; eqi$ to model the natural numbers and booleans together with a test function for equality with boolean values. So we want that

$$\begin{array}{llll} nat; bool & : Setoid & & \\ O & : nat & true; false & : bool \\ S & : nat \; ! \; nat & eq & : nat \quad nat \; ! \; bool \end{array}$$

So in this case $n = 2$, the index of the sort nat is 0, the index of the sort $bool$ is 1, and the types of constants and functions are

O	$)$	$h[]; 0i$	(no arguments and result in nat)
S	$)$	$h[0]; 0i$	(one argument from nat, result in nat)
$true$	$)$	$h[]; 1i$	(no arguments, result in $bool$)
$false$	$)$	$h[]; 1i$	(no arguments, result in $bool$)
eq	$)$	$h[0; 0]; 1i$	(two arguments from nat, result in $bool$)

Definition 9. *Let $n : \mathbb{N}$ be a fixed natural number. Let $Sort \quad \mathbb{N}_n$. A* function type *is a pair $hargs; resi$, where $args$ is a list of elements of $Sort$ (indicating the type of the arguments of the function) and res is an element of $Sort$ (indicating the type of the result). So in Type Theory we define the type of function types as $(Function\_type\ n) := (list\ Sort) \quad Sort:$*

Definition 10. *A* signature *is a pair $hn; fsi$ where $n : \mathbb{N}$ and $fs \quad [f_1; : : : ; f_m]$ is a list of function types. We represent it in Type Theory by a record type:*

$$Record\ Signature : \ ^s :=$$
$$signature \begin{array}{ll} sorts\_num & : \mathbb{N} \\ function\_types & : (list\ (Function\_type\ sorts\_num)) \end{array}$$

The signature of natural numbers and booleans is then defined as $=$ $(signature\ 2\ [h[]; 0i; h[0]; 0i; h[]; 1i; h[]; 1i; h[0; 0]; 1i])$.

Definition of algebra. Let $: Signature$, we want to define the notion of a -algebra. To define such a structure we need to interpret the sorts as setoids, and the function types as setoid functions. Suppose $= hn; [f_1; : : : ; f_s]i$. The interpretation of the sorts is a family of n setoids: $Sorts\_interpretation := \mathbb{N}_n \; ! \; Setoid$. So let us assume that $sorts : Sorts\_interpretation$, and define the interpretation of $f_1; : : : ; f_n$. There are several ways of defining the type of a function, depending on how the arguments are given. Suppose $f = h[a_1; : : : ; a_k]; ri$ is a function type. If $x_j : (sorts\ a_j)$ for $j = 1; : : : ; k$, then we would like the interpretation of f, kfk, to be applicable directly to its arguments, $(kfk\ x_1\ : : : x_k) : (sorts\ r)$. This means that kfk should have the curried type $(sorts\ a_1)[\ !\ ] : : : [\ !\ ](sorts\ a_k)[\ !\ ](sorts\ r)$ This type may be defined by using a general construction to define types of curried functions with arity and types of the arguments as parameters. This is done by the function

$$Curry\_type\_setoid : (n : nat)(\mathbb{N}_n \; ! \; Setoid) \; ! \; Setoid \; ! \; Setoid$$

such that if n is a natural number, $A : \mathbb{N}_n \to \mathit{Setoid}$ is a family of setoids defining the type of the arguments, and $B : \mathit{Setoid}$ is the type of the result, then

$$(\mathit{Curry\_type\_setoid}\ n\ A\ B) = (A\ 0)[\to]\ldots[\to](A\ n-1)[\to]B$$

So in the previous example the type of kfk may be defined as

$$(\mathit{Curry\_type\_setoid}\ k\ [i : \mathbb{N}_k](\mathit{sorts}\ a_i)\ (\mathit{sorts}\ r))$$

But this representation is difficult to use when reasoning abstractly about functions, e.g. if we want to prove general properties of the functions which do not depend on the arity. In this situation it is better to see the function having just one argument containing all the x_j's. We can do that by giving the arguments as k-tuples or as functions indexed on a finite type. We choose this second option. So we represent the arguments as an object of type $(j : \mathbb{N}_k)(\mathit{sorts}\ a_j)$. Then the interpretation of the function f could have the type $((j : \mathbb{N}_k)(\mathit{sorts}\ a_j)) \to (\mathit{sorts}\ r).$ This is still not completely correct. Since the sorts are setoids, the interpretation of the functions must preserve the setoid equality. With the aim of formulating this condition, we first make the type of arguments $(j : \mathbb{N}_k)(\mathit{sorts}\ a_j)$ into a setoid by stating that two elements $\mathit{args}_1, \mathit{args}_2$ are equal if they are extensionally equal.

Definition 11. *Let* $k : \mathbb{N}$, $A : \mathbb{N}_k \to \mathit{Setoid}$. *Then* $(\mathit{FF\_setoid}\ k\ A)$ *is the setoid that has carrier* $(j : \mathbb{N}_k)(A\ j)$ *and equality relation*

$$(\mathit{args}_1 =_{(\mathit{FF\_setoid}\ k\ A)} \mathit{args}_2) \iff (\forall j : \mathbb{N}_k)((\mathit{args}_1\ j) =_{(A\ j)} (\mathit{args}_2\ j))$$

We can now interpret a function type and a list of function types.

Definition 12. *Let* $f = \langle[a_1, \ldots, a_k], r\rangle$ *be a function type. Then*

$$\begin{array}{l}(\mathit{Function\_type\_interpretation}\ n\ \mathit{sorts}\ f) := \\ (\mathit{FF\_setoid}\ k\ [i : \mathbb{N}_k](\mathit{sorts}\ a_k))[\to](\mathit{sorts}\ r)\end{array}$$

A list of function types is interpreted by the operator

$$\begin{array}{l}(\mathit{Function\_list\_interpretation}\ n\ \mathit{sorts}) : \\ (\mathit{list}\ (\mathit{Function\_type}\ n)) \to \mathit{Setoid}\end{array}$$

where the carrier of $(\mathit{Function\_list\_interpretation}\ n\ \mathit{sorts}\ [f_1, \ldots, f_s])$ *is*

$$[i : \mathbb{N}_s](\mathit{Function\_type\_interpretation}\ n\ \mathit{sorts}\ f_i)$$

(we do not need to take into consideration how the equality relation is defined).

This is the way in which functions are represented in the algebra. Whenever we want to have them in the curried form we can apply a conversion operator

$$\mathit{fun\_arg\_to\_curry} : ((\mathit{FF\_setoid}\ k\ A)[\to]B) \to (\mathit{Curry\_type\_setoid}\ k\ A\ B).$$

The inverse conversion is performed by the operator $\mathit{curry\_to\_fun\_arg}$.

Eventually, the type of -algebras can be defined as

Definition 13. *The type of* algebras over the signature *is the record type*

$$\begin{array}{l} Record\ (Algebra\ \ ) : \Box := \\ \quad algebra \left\{ \begin{array}{ll} sorts & : (Sorts\_interpretation\ (sorts\_num\ \ )) \\ functions & : (Function\_list\_interpretation \\ & \quad (sorts\_num\ \ )\ sorts\ (function\_types\ \ )) \end{array} \right. \end{array}$$

The type of arguments corresponding to the i-th function of the signature in an algebra A are also indicated by $(Fun\_arg\_arguments\ A\ i)$.

If $\langle n;[f_0;\ldots;f_{m-1}]\rangle$ and $A : (Algebra\ \ )$, we indicate by f_{i_A} the interpretation of the ith function symbol f_{i_A} $(functions\ \ A\ i)$ for every $i : \mathbb{N}_m$. As an example let us define a -algebra for the signature considered before, interpreting the two sorts as the setoids of natural numbers and booleans (in these cases the equivalence relation is trivially Leibniz equality). Suppose we have already defined

$$\begin{array}{lll} Nat; Bool & : Setoid & \\ 0 & : Nat & T; F : Bool \\ S & : Nat[\rightarrow]Nat & Eq\ \ : Nat[\rightarrow]Nat[\rightarrow]Bool \end{array}$$

Then we can give the interpretation of the sorts

$$\begin{array}{l} Srt : (Sorts\_interpretation\ 2) \\ (Srt\ 0) = Nat \qquad (Srt\ 1) = Bool \end{array}$$

and of the functions

$$\begin{array}{ll} \multicolumn{2}{l}{Fun : (Function\_list\_interpretation\ Srt\ (function\_types\ sigma))} \\ (Fun\ 0) = (curry\_to\_fun\_arg\ 0) & (Fun\ 3) = (curry\_to\_fun\_arg\ F) \\ (Fun\ 1) = (curry\_to\_fun\_arg\ S) & (Fun\ 4) = (curry\_to\_fun\_arg\ Eq) \\ (Fun\ 2) = (curry\_to\_fun\_arg\ T) & \end{array}$$

Then we can define the -algebra

$$nat\_bool\_alg := (algebra\ \ Srt\ Fun) : (Algebra\ \ )$$

4 Term Algebras

Informal definition of term algebras. A class of algebras of special interest is that of Term Algebras. The sorts of such an algebra are the terms freely generated by the function symbols of the signature. For example, in the signature defined above we would have that the expressions O, $S(O)$, $S(S(O))$ are terms of the first sort, while $true$, $false$, $eq(O;S(O))$ are terms of the second. In general given a signature $= \langle n;[f_1;\ldots;f_m]\rangle$, the algebra of terms have carriers T_i, for $i : \mathbb{N}_n$, whose elements have the form $f_j(t_1;\ldots;t_k)$ where $j : \mathbb{N}_m$, the type of f_j is $\langle[a_1;\ldots;a_k];r\rangle$, $t_1;\ldots;t_k$ belong to the term sorts $T_{a_1};\ldots;T_{a_k}$ respectively, and the resulting term is in the sort T_r.

Similarly we can define an algebra of open terms or expressions, i.e. terms in which variables can appear. We start by a family of sets of variables X_i for $i : \mathbb{N}_n$, and we construct terms by application of the function symbols as before.

Problem: the uniform definition. In Type Theory this can be easily modeled by inductively defined types whose constructors correspond to the functions of the signature. For example, the sorts of terms of the previous signature are the (mutually) inductive types

$$\begin{array}{lll} nat\_term := & o\_symb & : nat\_term \\ & j\ s\_symb & : nat\_term\ \ !\ \ nat\_term \\ bool\_term := & t\_symb & : bool\_term \\ & j\ f\_symb & : bool\_term \\ & j\ eq\_symb & : nat\_term\ \ !\ \ nat\_term\ \ !\ \ bool\_term \end{array}$$

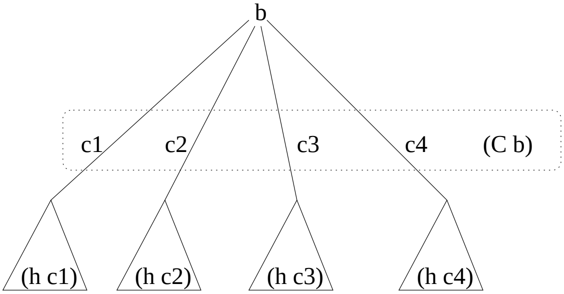
If the signature is single-sorted, a simple inductive definition gives the type of terms; if it is many-sorted then we have to use mutually inductive definitions. In this way we can define the types of sorts for any specific signature, but it is not possible to define it parametrically. We would like to define term algebras as a second order function

$$Term\_algebra : (\quad : Signature)(Algebra\quad)$$

that associates the corresponding term algebra to each signature. In order to do this we would need mutually inductive definitions in which the number of sorts and constructors and the type of the constructors are parametric. Such a general form of inductive definition is not available in current implementations of Type Theory (like Coq), so we have to look for a different solution.

Discussion on possible solutions. The problem is more general and regards the definition of families of inductive types in which every element of the family is a correct inductive type, but the family itself cannot be defined. In the general case we have a family of set operators indexed on a set A, $\quad : A\ \ !\ \ (\ ^s\ \ !\ \ ^s)$ and we want to define a family of inductive types each of which is the minimal fixed point of the corresponding operator, i.e. we want a family $I : A\ \ !\ \ ^s$ such that for every $a : A$, $(I\ a)$ is the minimal fixed point of $(\quad a)$. In Type Theory it is possible to define the minimal fixed point of a set operator $\Psi :\ ^s\ \ !\ \ ^s$ if and only if the set operator is strictly positive, i.e. in the expression $(\Psi\ X)$, where $X :\ ^s$, X occurs only to the right of arrows. But it may happen that even if for every concrete element (closed term) a of the set A, the operator $(\quad a)$ is strictly positive or reduces to a strictly positive operator, this does not hold for open terms, i.e. if $x : A$ is a variable $(\quad x)$ does not satisfy the strict positivity condition. There are several possibilities to overcome this difficulty. A thorough analysis of this subject will be the argument of a future paper. Here we adopt a solution that represents every inductive type by a type of trees.

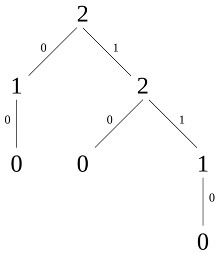
Solution using Wellorderings. W types are a type theoretic implementation or the notion of well orderings as well-founded trees. They were introduced by Per Martin-Löf in [16] (see also [17] and [19], chapter 15). Suppose that we want to define a type of trees such that the nodes of the trees are labeled by elements of the type B, and for each node labeled by an element $b : B$, the branches stemming from the node are labeled by the elements of a set $(C\ b)$, i.e. the b-node has as many branches as the elements of $(C\ b)$. Then the W type constructor has two parameters: a type $B : \ ^s$ and a family of types $C : B \ ! \ \ ^s$. To define a new element of the type $(W\ B\ C)$ we have to specify the label of the root by an element $b : B$ and for each branch, i.e. for every element $c : (C\ b)$, the corresponding subtree; this is done by giving a function $h : (C\ b) \ ! \ (W\ B\ C)$.



Formally we can define $(W\ B\ C)$ in the Calculus of Inductive Constructions (see [7] and [9]) as the inductive type $(W\ B\ C)$ with one constructor sup : $(b : B)((C\ b) \ ! \ (W\ B\ C)) \ ! \ (W\ B\ C)$. As for any inductive definition, we automatically get principles of recursion and induction associated with the definition. If $(C\ b)$ is infinite for some $b : B$ we get transfinite induction.

We can use this construction to define term algebras for single-sorted signatures, representing a term by its syntax tree. We choose B to be the set of function symbols of the signature (or just $\mathbb{N}_m$ where m is the number of the functions), and $(C\ f) = \mathbb{N}_{k_f}$ where k_f is the arity (number of arguments) of the function symbol f.

For example, let us take the signature $h1; [h[]; 0i; h[0]; 0i; h[0; 0]; 0i]i$ describing a structure with one sort, one constant, one unary operation and one binary operation. Let us indicate the three functions by f_0 (the constant), f_1 (the unary operation) and f_2 (the binary operation). The type of terms is represented by the type $(W\ \mathbb{N}_3\ C)$ where $C = [i : \mathbb{N}_3](cases\ i\ of\ 0\ )\ \mathbb{N}_0 j 1\ )\ \mathbb{N}_1 j 2\ )\ \mathbb{N}_2):$ Then the term $f_2(f_1(f_0); f_2(f_0; (f_1(f_0))))$ is represented by the tree



or formally by the element of $(W\ \mathbb{N}_3\ C)$

$$\begin{array}{l}
(\mathit{sup}\ 2\ [i:\mathbb{N}_2](\mathit{cases}\ i\ \mathit{of} \\
\qquad 0\ )\ \ (\mathit{sup}\ 1\ [j:\mathbb{N}_1](\mathit{cases}\ j\ \mathit{of}\ 0\ )\ \ (\mathit{sup}\ 0\ [k:\mathbb{N}_0](\mathit{cases}\ k\ \mathit{of})))) \\
\quad\ \ j\,1\ ) \\
\qquad\qquad (\mathit{sup}\ 2\ [l:\mathbb{N}_2](\mathit{cases}\ l\ \mathit{of} \\
\qquad\qquad\qquad 0\ )\ \ (\mathit{sup}\ 0\ [k:\mathbb{N}_0](\mathit{cases}\ k\ \mathit{of})) \\
\qquad\qquad\quad\ \ j\,1\ )\ \ (\mathit{sup}\ 1\ [j:\mathbb{N}_1](\mathit{cases}\ j\ \mathit{of} \\
\qquad\qquad\qquad\qquad 0\ )\ \ (\mathit{sup}\ 0\ [k:\mathbb{N}_0](\mathit{cases}\ k\ \mathit{of})))))))
\end{array}$$

(Of course, for practical uses we have to define some syntactic tools, to spare the user the pain of writing such terms.)

General Tree Types. To deal with multi-sorted signatures we need to generalize the construction. The General Trees type constructor that we use is very similar to that introduced by Kent Petersson and Dan Synek in [20] (see also [19], chapter 16).

In the multi-sorted case we have to define not just one type of terms, but n types, if n is the number of sorts. These types are mutually inductive. So we define a family $\mathbb{N}_n\ \ !\ \ {}^s$. In general we consider the case in which we want to define a family of tree types indexed on a given type A, so the elements of A are though of as indexes for the sorts. For what regards the functions, besides their arity we have to take into account from which sort each argument comes and to which sort the result belongs. Like before we have a type B of indexes for the functions. To each $b : B$ we have to associate, as before, a set $(C\ b)$ indexing its arguments. But now we must also specify the type of the arguments: to each $c : (C\ b)$ we must associate a sort index (the sort of the corresponding argument) $(g\ b\ c) : A$. Therefore we need a function $g : (b : B)(C\ b)\ \ !\ \ A$. Furthermore we must specify to which sort the result of the application of the function b belongs, so we need an other function $f : B\ \ !\ \ A$. Then in the context

$$\begin{array}{ll}
A;B:\ {}^s & \qquad f:B\ \ !\ \ A \\
C\quad :B\ \ !\ \ {}^s & \qquad g:(b:B)(C\ b)\ \ !\ \ A
\end{array}$$

we define the inductive family of types $(\mathit{General\_tree}\ A\ B\ C\ f\ g) : A\ \ !\ \ {}^s$ with the constructor (we write Gt for $(\mathit{General\_tree}\ A\ B\ C\ f\ g)$)

$$\mathit{g\_tree} : (b:B)((c:(C\ b))(\mathit{Gt}\ (g\ b\ c)))\ \ !\ \ (\mathit{Gt}\ (f\ b)))$$

In the case of a signature $\ = hn;[f_1;:::;f_m]i$ such that for every $i : \mathbb{N}_m$, $f_i = h[a_{i;0};:::;a_{i;k_i-1}];r_i i$ (k_i is the arity of f_i), we have

$$\begin{array}{lll}
A & = \mathbb{N}_n & \qquad\qquad f = [b:B]r_b \\
B & = \mathbb{N}_m & \qquad\qquad g = [b:B][c:(C\ b)]a_{b;c} \\
(C\ b) & = \mathbb{N}_{k_b}\quad \text{for every } b:B &
\end{array}$$

The family of types of terms is $(\mathit{Term}\quad) := (\mathit{General\_tree}\ A\ B\ C\ f\ g)$.

The problem of intensionality. One problem that arises when using the General Trees constructor to define term algebras is the intensionality of equality. A term (tree) is defined by giving a constructor $b : B$ and a function

$h : (c : (C\ b))(Term\ \ (g\ b\ c))$. It is possible that two functions $h_1; h_2 : (c : (C\ b))(Term\ \ (g\ b\ c))$ are extensionally equal, i.e. $(h_1\ c) = (h_2\ c)$ for all $c : (C\ b)$, but not intensionally equal, i.e. not convertible. In this case the two trees $(g\_tree\ b\ h_1)$ and $(g\_tree\ b\ h_2)$ are intensionally distinct. But we want two terms obtained by applying the same function to equal arguments to be equal. Since algebras are required to be setoids, and not just sets, we can solve this problem by defining an inductive equivalence relation on the types of terms that captures this extensionality,

$$\begin{array}{l} Inductive\ tree\_eq : (a : A)(Term\ \ a)\ !\ (Term\ \ a)\ !\ \ ^{p} := \\ \qquad tree\_eq\_intro : (b : B)(h_1; h_2 : (c : (C\ b))(Term\ \ (g\ b\ c))) \\ \qquad\qquad\qquad\qquad ((8c : (C\ b))(tree\_eq\ (g\ b\ c)\ (h_1\ c)\ (h_2\ c)))\ \ ! \\ \qquad\qquad\qquad\qquad (tree\_eq\ (f\ \ b)\ (g\_tree\ b\ h_1)\ (g\_tree\ b\ h_2)) \end{array}$$

Now we would like to prove that $(tree\_eq\ \ a)$ is an equivalence relation on $(Tree\ \ \ a)$. Unfortunately no proof of transitivity could be found. The problem can be formulated and generalized in the following way. If we have an inductive family of types and a generic element of one of the types in the family, it is not generally possible to prove an inversion result stating that the form of the element is an application of one of the constructors corresponding to that type. This was proved for the first time by Hofmann and Streicher in the case of the equality types (see [11]). Therefore we just took the transitive closure of the above relation.

Functions. We have constructed an interpretation of the sorts of a signature in setoids of terms as syntax trees. We still have to interpret the functions. This is not difficult given the way we defined the function interpretation. The functions are associated to the elements of the type $B = \mathbb{N}_m$. Given an element $b : B$ we have a function

$$(g\_tree\ b) : ((c : (C\ b))(Term\ \ \ (g\ b\ c)))\ \ !\ \ (Term\ \ \ (f\ \ b))$$

It is straightforward to prove that it preserves the setoid equality, so it has the right type to be the interpretation of the function symbol b. Let us call $(functions\_interpretation\ \ \ )$ this family of setoid functions. We then obtain the algebra of terms

$$\begin{array}{l} Term\_algebra\ \ \ \ : (\ \ : Signature)(Algebra\ \ \ ) \\ (Term\_algebra\ \ \ ) = (algebra\ \ \ (Term\ \ \ )\ (functions\_interpretation\ \ \ )): \end{array}$$

Expressions Algebras. We defined algebras whose elements are closed terms constructed from the function symbols of the signature . It is also very important to have algebras of open terms, or expressions, where free variables can appear. To do this we modify the definition of term algebras allowing a set of variables alongside that of functions. So in the construction of syntax trees some leaves may consist of variable occurrences. We assume that every sort has a countably infinite number of variables, so the set of variables is $Var := \mathbb{N}_n\ \ \ \ \mathbb{N}$. Then a variable is a pair $hs; ni$ where s determines the sorts to which it belongs and n says that it is the n-th variable of that sort. Variables are treated as constants, i.e. as function symbols of zero arity. In the definition of the term algebra we modify the set B of constructors: $B := \mathbb{N}_m + Var$ and also the family C giving the types of the subtrees in such a way that $(C\ (inr\ v))$ is the empty set for every variable v, while $(C\ (inl\ j)) = \mathbb{N}_{k_j}$ as before. The rest of the definition remains the same. We may also abstract from the actual set of variables and use any family $X : \mathbb{N}_n \to {}^{s}$ as the family of sets of variables. The closed terms are then a particular case obtained by taking $(X\ i) \quad \emptyset$ for every $i : \mathbb{N}_n$, and the previous case is obtained by taking $(X\ i) \quad \mathbb{N}$.

5 Congruences, Quotients, Subalgebras, and Homomorphisms

Congruences and quotients. If is a signature and A a -algebra, then we call congruence a family of equivalence relations over the sorts of A that is consistent with the operations of the algebra, i.e. when we apply one of the operations to arguments that are in relation then we obtain results that are in relation. Such condition is rendered in Type Theory by the following

Definition 14. *Let* $\langle n, [f_0, \ldots, f_{m-1}]\rangle : Signature$ *with* $f_i \quad \langle [a_{i,0}, \ldots, a_{i,h_i}], r_i\rangle$ *for* $i : \mathbb{N}_m$*, and* $A : (Algebra\ \ )$*. A family of relations on the sorts* $A = (sorts\ A)$*,* $(\ {}_s) : (Setoid\_relation\ (A\ s))$ *for* $s : \mathbb{N}_n$*, satisfies the* substitutivity condition $(Substitutivity\ (\ \ ))$ *if and only if*

$$(\forall i : \mathbb{N}_m)(\forall args_1, args_2 : (Fun\_arg\_arguments\ A\ i))$$
$$((\forall j : \mathbb{N}_{h_i})(args_1\ j)\ {}_{a_{i,j}}\ (args_2\ j)) \to (f_{i\,A}\ args_1)\ {}_{r_i}\ (f_{i\,A}\ args_2).$$

The type of congruences *over a -algebra* A *is the record type*

$$Record\ Congruence\ \ : \square := \\ congruence \begin{cases} cong\_relation : (s : \mathbb{N}_n)(Setoid\_relation\ (A\ s)) \\ cong\_equiv \quad : (s : \mathbb{N}_n)(Equiv\ (cong\_relation\ s)) \\ cong\_subst \quad : (Substitutivity\ cong\_relation) \end{cases}$$

So a congruence on A has the form $(congruence\ rel\ eqv\ sbs)$ where rel is a family of setoid relations on the sorts of A, eqv is a proof that every element of the family is an equivalence relation and sbs is a proof that the family satisfies the substitutivity condition.

Given a congruence over an algebra we can construct the quotient algebra. In classic Universal Algebra this is done by taking as sorts the sets of equivalence classes with respect to the congruence. In Type Theory, as we have already said about quotients of setoids, the quotient has exactly the same carriers, but we replace the equality relation. The substitutivity condition guarantees that what we obtain is still an algebra.

Lemma 1. *Let* $: Signature$*,* $A : (Algebra\ \ )$ *and* $(\ \ ) : (Congruence\ \ A)$*. If we consider the family of setoids obtained by replacing each* $=_{(sorts\ A\ s)}$ *by* ${}_s$ *the functions of* A *are still well defined. We can therefore define the* quotient algebra $A/\ \ $.

Subalgebras. The definition of subalgebra can be given in the same spirit of the definition of quotient algebras.

Definition 15. *Let* A : ($Algebra\ \ $) *and* P_s : ($Setoid\_predicate$ ($sorts\ A\ s$)) *a family of predicates on the sorts of* A. *We say that* P *is* closed under the functions *of* A *if*

$$(8i : \mathbb{N}_m)(8args : (Fun\_arg\_arguments\ A\ i))$$
$$((8j : \mathbb{N}_{h_i})(P_{a_j}\ (args\ j)))\ !\ (P_{r_j}\ (f_{i_A}\ args)):$$

Definition 16. *The* subalgebra $Aj\ P$ *is the* -*algebra with sorts* ($sorts\ A\ s)jP_s$ *and functions the restrictions of the functions of* A.

Notice that the restrictions of the functions of A to the subsetoids ($sorts$ $A\ s)jP_s$ are well-defined because P is closed under function application. The proof of this fact gives the proof of (P_{r_j} ($f_{i_A}\ args$)) and therefore allows the construction of a well-typed element of the -type which is the carrier of the subsetoid.

Homomorphisms. Given a signature : $Signature$ and two -algebras A and B, we want to define the notion of homomorphism between A and B. Informally an homomorphism is a family of functions $_s$: ($A\ s$) $!$ ($B\ s$), where s : $\mathbb{N}_n$ and A and B are the families of sorts of A and B respectively, that commutes with the interpretation of the functions of . That means that if f is one of the function types of and $a_1; :::; a_k$ are elements of the algebra A, belonging to the sorts prescribed by the types of the arguments of f, then, suppressing the index i in $_i$, (($kfk_A\ a_1\ :::\ a_k$)) = (kfk_B (a_1) $:::$ (a_k)) where kfk_A indicates the curried version of the interpretation of the function type f in the algebra A.

Formally we have first of all to require that is a family of setoid functions : ($i : \mathbb{N}_n$)($A\ i$)[$!$]($B\ i$)*:* Then the requirement that must commute with the functions of the signature must take into account the way we interpreted the function symbols. Let $i : \mathbb{N}_m$ be a function index, and $f_i = h[a_{i;0}; :::; a_{i;k_i-1}]; r_i i$ be the corresponding function type of . Assume we have an argument function for f_{i_A}, $args_A$: ($Fun\_arg\_arguments\ A\ i$). Remember that this is a function that to every j : $\mathbb{N}_{k_i}$ assign an element ($args_A\ j$) : ($sorts\ A\ a_{i;j}$). Then by applying to each argument we obtain an argument function for f_{i_B}, $args_B$:= [j : $\mathbb{N}_{k_i}$]($_{a_{i;j}}$ ($args_A\ j$))) : ($Fun\_arg\_arguments\ B\ i$). For to be an homomorphism we must then require that for every function index i the equality ($_{r_i}$ ($f_{i_A}\ args_A$)) $=_{(B\ r_i)}$ ($f_{i_B}\ args_B$) holds. Let us call this property ($Is_homomorphism\ \ $). Then we can define the type of homomorphisms as the record

$$Record\ Homomorphism :\ {}^{s} :=$$
$$homomorphism\ \begin{array}{ll} hom\_function & : (i : \mathbb{N}_n)(A\ i)[\ !\ ](B\ i) \\ hom\_proof & : (Is\_homomorphism\ \ ) \end{array}$$

By requiring that the setoid functions $_i$ are injective, surjective or bijective we get respectively the notions of monomorphism, epimorphism and isomorphism. We also call endomorphisms (automorphisms) the homomorphisms (isomorphisms) from an algebra A to itself.

Term evaluation. One important homomorphism is the one from the term algebra $T = (\mathit{Term\_algebra}\ )$ to any -algebra A. This homomorphism is unique since the interpretation of all terms is determined by the interpretation of functions. $\mathit{term\_evaluation}$ can be defined by induction on the tree structure of terms in such a way that $(\mathit{term\_evaluation}\ (f_{i_T}\ \mathit{args})) = (f_{i_A}\ \mathit{args}^0)$ where $\mathit{args}^0 := [j : \mathbb{N}_{k_i}](\mathit{term\_evaluation}\ (\mathit{args}\ j))$ and we have suppressed the sort indexes. After proving that $\mathit{term\_evaluation}$ is a setoid function (preserves the equality of terms) and that it commutes with the operations of , we obtain an homomorphism $\mathit{term\_ev} : (\mathit{Homomorphism}\ \ T\ A)$.

Similarly we can define the evaluation of expressions containing free variables. In this case the function $\mathit{expression\_evaluation}$ takes an additional argument $\mathit{ass} : (\mathit{Assignment}\ \ A)$ assigning a value in the right sort of A to every variable: $(\mathit{Assignment}\ \ A) := (v : (\mathit{Var}\ \ ))(A\ (\ _1\ v))$. Using this extra argument to evaluate the variables, we can construct as before an homomorphism $\mathit{expression\_ev} : (\mathit{Homomorphism}\ \ E\ A)$ where $E = (\mathit{Expressions\_algebra}\ \ )$.

Kernel of a homomorphism. Associated to every homomorphism of -algebra $: (\mathit{Homomorphism}\ \ A\ B)$ there is a congruence on A called the *kernel* of .

Definition 17. *The* kernel *of the homomorphism is family of relations*

$$\begin{array}{ll} (\mathit{ker\_rel}\ \ ) & : (s : \mathbb{N})(\mathit{relation}\ (A\ s)) \\ (\mathit{ker\_rel}\ \ s\ a_1\ a_2) & (\ )\ \ (\ _s\ a_1) =_{(B\ s)} (\ _s\ a_2) \end{array}$$

Lemma 2. $\mathit{ker\_rel}$ *is a congruence on* A.

The kernel of is indicated by the standard notation .

We can, therefore, take the quotient $A=$ and consider the natural homomorphism between A and $A=$. In classic Universal Algebra this homomorphism associates to every element a in A the equivalence class $[a]$. But in our implementation the carriers of A and $A=$ are the same and so the natural homomorphism is just the identity. We only have to verify that it is actually an homomorphism (it preserves the setoid equality and it commutes with the operation of the signature).

Lemma 3. *For any -algebra A and any congruence on A, the family of identity functions $[s : \mathbb{N}_n][x : (\mathit{sorts}\ \ A\ s)]x$ is a homomorphism from A to $A=$.*

In the case of such homomorphism is indicated by nat .

First homomorphism theorem. Once we have developed the fundamental notions of Universal Algebra in Type Theory and we have constructed operators to manipulate them, we can prove some standard basic results like the following.

Theorem 1 (First Homomorphism Theorem). *Let A and B be two -algebras and : ($Epimorphism\ \ A\ B$). Then there exists an isomorphism*

$$(ker\_quot\_iso\ \ ) : (Isomorphism\ \ A{=}\ \ \ B)$$

such that (ker_quot_iso) nat = , where the equality is the extensional functional equality.

6 Conclusions and Further Research

We have implemented in Type Theory (using the proof development system Coq for the formalization) the fundamental notions and results of Universal Algebras. This implementation allows us to specify any first order algebraic structure and has operators to construct free algebras over a signature. We defined the constructions of subalgebras, product algebras and quotient algebras and proved their basic properties. There were two main points in which we had to employ special type theoretic constructions: we used setoids as carriers for algebras in order to be able to define quotient algebras and we used wellorderings to represent free algebras.

This implementation is intended to serve two purposes. From the practical point of view it provides a set of tools that make the use of Type Theory in the development of mathematical structures easier. From the theoretical point of view it investigates the use of Type Theory as a foundation for Mathematics.

An important line of research that we intend to pursue is that of equational reasoning. The next steps in this direction are the definition of equational classes of algebras, where equations are represented by pairs of open terms, and the proof of Birkhoff's soundness theorem. This will give us tools to automatically prove formulas over a generic algebra by lifting them to the syntactic level of expressions.

References

[1] Peter Aczel. Notes towards a formalisation of constructive galois theory. draft report, 1994.

[2] H. P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*. Oxford University Press, 1992.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Henri Laulhere, Cesar Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Sa¨bi, and Benjanin Werner. *The Coq Proof Assistant Reference Manual. Version 6.2.*

[4] G. Barthe, M. Ruys, and H. P. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume LNCS 1158, pages 16{35. Springer, 1995.

[5] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Mart n Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume LNCS 1281, pages 515{529. Springer, 1997.

[6] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[7] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf, editor, *Proceedings of Colog '88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[8] Thierry Coquand and Henrik Persson. Integrated Development of Algebra in Type Theory. Presented at the Calculemus and Types '98 workshop, 1998.

[9] Eduardo Gimenez. A Tutorial on Recursive Types in Coq. Technical report, Unite de recherche INRIA Rocquencourt, 1998.

[10] Martin Hofmann. Elimination of extensionality in Martin-Löf type theory. In Barendregt and Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES '93*, pages 166{190. Springer-Verlag, 1993.

[11] Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 208{212. IEEE Computer Society Press, 1994.

[12] Douglas J. Howe. Computational metatheory in Nuprl. In E.Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume LNCS 310, pages 238{257. Springer-Verlag, 1988.

[13] Gerard Huet and Amokrane Sa¨bi. Constructive category theory. In *In honor of Robin Milner*. Cambridge University Press, 1997.

[14] Paul Jackson. Exploring abstract algebra in constructive type theory. In *Automated Deduction { CADE-12*, volume Lectures Notes in Artificial Intelligence 814, pages 591{604. Springer-Verlag, 1994.

[15] Zhaohui Luo. *Computation and Reasoning, A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

[16] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153{175. North-Holland, 1982.

[17] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[18] K. Meinke and J. V. Tucker. Universal Algebra. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 1*. Oxford University Press, 1992.

[19] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Clarendon Press, 1990.

[20] Kent Petersson and Dan Synek. A Set Constructor for Inductive Sets in Martin-Löf's Type Theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, U.K.*, volume 389 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[21] Amokrane Sa¨bi. Typing algorithm in type theory with inheritance. In *POPL'97: The 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 292{301. Association for Computing Machinery, 1997.

[22] Milena Stefanova. *Properties of Typing Systems*. PhD thesis, Computer Science Institute, University of Nijmegen, 1999.

[23] Laurent Thery. Proving and Computing: a certified version of the Buchberger's algorithm. Technical report, INRIA, 1997.

Locales
A Sectioning Concept for Isabelle

Florian Kammüller[1], Markus Wenzel[2], and Lawrence C. Paulson[1]

[1] Computer Laboratory, University of Cambridge
[2] Technische Universität München, Institut für Informatik

Abstract. Locales are a means to define local scopes for the interactive proving process of the theorem prover Isabelle. They delimit a range in which fixed assumption are made, and theorems are proved that depend on these assumptions. A locale may also contain constants defined locally and associated with pretty printing syntax.
Locales can be seen as a simple form of modules. They are similar to sections as in AUTOMATH or Coq. Locales are used to enhance abstract reasoning and similar applications of theorem provers. This paper motivates the concept of locales by examples from abstract algebraic reasoning. It also discusses some implementation issues.

1 Introduction

In interactive theorem proving it is desirable to get as close as possible to the convenience of paper proof style, making developments more comprehensible and self declaring. In mathematical reasoning, assumptions and definitions are handled in a casual way. That is, a typical mathematical proof assumes propositions for one proof or a whole section of proofs and local to these assumption definitions are made that depend on those assumptions. The present paper introduces a concept of *locales* for Isabelle [Pau94] that aims to support the described processes of local assumptions and definition. Locales are implemented and have been first released with Isabelle98-1.

In mathematical proofs, we often want to define abbreviations for big expressions to enhance readability. These abbreviations might implicitly refer to terms which are arbitrary but fixed values for the entire proof. Isabelle's pretty printing and definition possibilities are mostly sufficient for this purpose. But there are still examples where a definition in a theory is too strong in the sense that the syntactical constants used for abbreviations are of no global significance. Definitions in an Isabelle theory are visible everywhere.

In the case study of Sylow's theorem [KP99], we came across several such local definitions. There, we define a set M as $fS \quad G{:}hcri \ j \ order(S) = p \quad g$ where G, p, and are arbitrary but fixed values with certain properties. This is just for one single big proof, and has no general purpose whatsoever. The formula does not even occur in the main proposition. Still, in Isabelle98 as it is, we only have the choice of spelling this term out wherever it occurs, or defining it on

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 149{165, 1999.

the global level, which is rather unnatural. Then we would have to parameterize over all variables of the right hand side. In our example we would get something like $M(G;p; \ )$ which is almost as bad as the original formula.

1.1 Related Work

There are several theorem provers that support modules, *e.g.* IMPS [FGT93], PVS [OSRSC98, ORR$^+$96], and Larch [GH93]. The authors of these systems suggest to use their modules for the representation of mathematical structures, for example abstract algebraic structures like groups. This representation by modules is often not adequate because the modules have no representation in the logic. The \little theories" of IMPS come closest to an adequate representation of mathematical structures by providing a transformation between types and sets (cf. [Kam99a, Chapt. 2]).

However, modules offer locality by providing local contexts in which formulas can make use of local declarations and definitions. Locales provide the locality that is part of a module concept. For adequate representation of mathematical structures we propose the concept of and -types as it is common in type theories. The first author adapted this approach for Isabelle/HOL [Kam99b] set theory. In general, type theories are more suited for the adequate representation of mathematical structure than classical logics. But, not everyone wants to use type theory.

Locales implement a sectioning device similar to that in AUTOMATH [dB80] or Coq [Dow90, CCF$^+$96]. In contrast to this kind of sections, locales are defined statically. Also, optional pretty printing syntax and dependent local definitions are part of the concept. Windley [Win93] describes abstract theories for HOL [GM93]. They are more adequate than classical modules, but do not offer the same notational advantages as locales. Deviating from the other approaches, locales do not have an instantiation mechanism, instead they enable exporting of theorems for individual instantiation (cf. Sect. 3.2).

1.2 Overview

Subsequently, we explain a simple approach to sectioning for the theorem prover Isabelle. In Sect. 2 we describe the locale concept and address issues of opening and closing of locales. We present aspects concerning concrete syntax, including a means for local definitions. We continue in Sect. 3 with the fundamental operations on locales and their features. Section 4 describes the implementation of the ideas. We give a simple example illustrating an application from algebra in Sect. 5. Finally, we discuss more general aspects of locales in Sect. 6.

2 Locales { The Concept

Locales delimit a scope of *locally fixed variables*, *local assumptions*, and *local definitions*. Theorems that are proved in the context of locales may make use of these local entities. The result will then depend on the additional hypotheses, while proper local definitions are eliminated.

A locale consists of a set of constants (with optional pretty printing syntax), rules and definitions. Defined as named objects of an Isabelle theory, locales can be invoked later in any proof session. By virtue of such an invocation, any locale rules and definitions are turned into theorems that may be applied in proof procedures like any other theorem. Similarly, the definitions may abbreviate longer terms, just like ordinary Isabelle definitions. However, the rules and definitions are only local to the scope that is defined by a locale.

Theorems proved in the scope of a locale can be exported to the surrounding theory context. In that case, rules employed for the proof become *meta-level assumptions* of the exported theorem. For the case of actual definitions, these hypotheses are eliminated via generalization and reflexivity. Thus the proof result becomes an ordinary theorem of the enclosing Isabelle theory.

Subsequently, we explain several aspects of locales. There are basically two ideas that form the concept of locales: one is the possibility to state local assumptions, and the other one is to make local definitions which can depend on these assumptions, and may use pretty printing. With those two main ideas the notion of a locale constant is strongly connected.

2.1 Locale Rules

To explain what locales are it is best to describe the main characteristics of Isabelle that lead to this concept and are the basis of their realization. The feature of Isabelle that builds the basis for the locale rules is Isabelle's concept of meta-assumptions.

In Isabelle each theorem may depend upon meta-assumptions. They are the antecedents of theorems of Isabelle's meta-logic — a form of the predicate calculus defined by a system of natural deduction rules. Meta-assumptions usually remain fixed throughout a proof and may be used within it in any order. The judgment that ψ holds under the meta-assumptions $\phi_1, \ldots, \phi_n$ is written as

$$\psi\ [\phi_1; \ldots; \phi_n]$$

A more conventional notation for this would be $\phi_1, \ldots, \phi_n \vdash \psi$. Note that this implicit $\vdash$ is different from the implication of the meta-logic ==> (cf. Sect. 5).

The first main aspect of locales is to build up a local scope, in which a set of rules, the locale rules, are valid. The local rules are realized by using Isabelle's meta-assumptions as an assumption stack. Logically, a locale is a conjunction of meta-assumptions; the conjuncts are the locale rules. Opening the locale corresponds to assuming this conjunction.

In Isabelle98, a meta-assumption can be introduced in proofs at any time, but by the end of the proof, Isabelle would complain about extraneous hypotheses. From Isabelle98-1 onwards, when the locale concept has been added, locale rules become meta-assumptions when the locale is invoked. A theorem proved in the scope of some locale, can use these rules. The result extraction process at the end of a proof has been modified accordingly to cope with this: the additional premises stemming from the locale are entailed in the conjunction; the proof result is admitted with the additional premises attached as meta-assumptions of the theorem. Hence, if this theorem is used in the same locale, the locale rules will be matched automatically, instead of producing new subgoals. All locale rules can be used throughout the life time of the locale. The life time is determined by the interactive operations of opening and closing (cf. Sect. 3.2).

2.2 Locale Constants

There is a notion of a *locale constant* that is integral part of the locale concept. A locale implements the idea of “arbitrary but fixed" that is used in mathematical proofs. We can assume certain terms as fixed for a certain section of proofs, and we can state further rules or define other terms depending on them. These arbitrary but fixed terms are the locale constants. The locale constants may be viewed from the outside as parameters, because they become universally quantified variables, when a result theorem is exported.

The idea of the locale constant is to use the locale as a scope such that inside the locale a free variable can be considered as a constant. Technically, locale constants behave like logical constants while the locale is open. In particular, they may be subject to the standard Isabelle pretty printing scheme, e.g. equipped with infix syntax.

A locale corresponds to a certain extent to modules in a theorem prover, with some notable restrictions of declaring items, though. In particular, a locale may not contain type constructor declarations and the constants are not persistent. The outside view of locales is realized in a different way. Instead of presenting the entire locale similar to a parameterized module that can be instantiated, one can export theorems from inside the locale. This export transforms a theorem into a general form whereby the locale is represented in the assumptions of the theorem.

2.3 Local Definition and Pretty Printing

A major reason for having a sectioning device like locales are user requirements to make temporary abbreviations in the course of a proof development. As pointed out in Sect. 1, there are large formulas that are used in proofs and do not have a global significance. Moreover, they might not even occur in the final conjecture of the theorem that we want to use. Conceptually, the definition of such logical terms is not a persistent definition. Nevertheless, we want to use such definitions to make the theorems readable, and the proof process clear. Hence, one aspect is the locality of these definitions. The other aspect, as illustrated by the introductory example as well, is that the local definitions might depend on terms that are constants in a certain scope. For example, we want to write M only, not a notation like $M(G;p;\ )$ as it would be necessary, if we wanted to refer to the terms that form the other premises in this particular proof [KP99].

Another common thing in abstract algebra are formulas which are not so big, but suppress implicit information, e.g. we write Ha for the right coset of a with respect to the subset H of a group G. Since the group G containing this coset is a parameter to this definition we would have to define something like r_coset G H a. This is partly the same problem as with the parameters of the definition M. Note that the normal pretty printing mechanism would not solve this problem either: neither definitions nor pretty printing syntax can hide arguments, like G here, although these are fixed in a local context.

These features are realized by locales. In a locale where G is an arbitrary but fixed group for a series of theorems we can use a syntax like H #> a instead of r_coset G H a. We create a simple *locale definition* mechanism for concrete syntax which implements the concept of a local definition with optional pretty printing syntax. The concept of such local definitions is based on the locale constant: inside a locale, a locale constant can be used to abbreviate longer terms. The terms we define can even be dependent on other locale constants if those are contained in the scope of the locale. Since locale constants are only temporarily fixed the latter feature realizes dependent definitions, i.e. the defined terms may omit implicit information of the context. This concrete syntax may only be used as long as the locale is open. Viewed from outside the locale, this syntax does not exist. The theorems proved inside the locale using the syntax can be transformed into global theorems with the syntactical abbreviations unfolded and the locale constants replaced by free variables.

In a locale where we want to reason about a group G and its right cosets, we declare G as a locale constant. Then we can define another locale constant #>, and define this in terms of the underlying theory of groups where the operation r_coset is defined generally.

```
rcos_def  "H #> x == r_coset G H x"
```

If the locale containing this definition is open, we can use the convenient syntax H #> x for right cosets, and it is defined as the sound operation of right cosets with the parameter G fixed for the current scope. If we finish a theorem and want to use it as a general result, we can *export* it. Then, the locale constant G will be turned into a universally quantified variable, and the definition will be expanded to the underlying adequate definition of right cosets.

3 Operations on Locales

Locales are introduced as named syntactic objects within Isabelle theories. They can then be opened in any theory that contains the theory they are defined for.

3.1 Defining Locales

The ideas of locale definitions, rules, and constants can be combined together to realize a sectioning concept. Thereby, we attain a mechanism that constitutes a local theory mechanism. To adjust this rather dynamic idea of definition and declaration to the declarative style of Isabelle's theory mechanism, we integrate the definition of locales into the theories as another language element of Isabelle theory files. The concrete syntax of locale definitions is demonstrated by example below. Locale group assumes the definition of groups as a set of records [NW98, Kam99b] as follows (cf. Sect. 5).

```
locale group =
  fixes
    G         :: "'a grouptype"
    e         :: "'a"
    binop     :: "'a => 'a => 'a"        (infixr "#" 80)
    inv       :: "'a => 'a"              ("i (_)" [90] 91)
  assumes
    Group_G   "G : Group"
  defines
    e_def     "e == (G.<e>)"
    binop_def "x # y == (G.<f>) x y"
    inv_def   "i x == (G.<inv>) x"
```

The above part of an Isabelle theory file introduces a locale for abstract reasoning about groups.

The subsection introduced by the keyword fixes declares the locale constants in a way that closely resembles Isabelle's global consts declaration. In particular, there may be an optional pretty printing syntax for the locale constants. As illustrated in the example, the user can define syntactical notations for operators, by defining a pattern for the application, as for the prefix syntax of inv. Alternatively, one can use the keywords infixr or infixl, as in the example of binop, to define a right or left associative infix syntax.

The subsequent assumes specifies the locale rules. They are defined like Isabelle rules, i.e. by an identifier followed by the rule given as a string. Locale rules admit the statement of local assumptions about the locale constants. The assumes part is optional. Non-fixed variables in locale rules are automatically bound by the universal quantifier !! of the meta-logic. In the above example, we assume that the locale constant G is a member of the set Group, i.e. is a group.

Finally, the defines part of the locale introduces the definitions that shall be available in this locale. Here, locale constants declared in the fixes section can be defined using the Isabelle meta-equality ==. The definition can contain variables on the left hand side, if the defined locale constant has appropriate type. This improves natural style of definition, for example for constants that represent infix operators, e.g. binop. The non-fixed variables on the left hand side are considered as schematic variables and are bound automatically by universal quantification of the meta-logic. The right hand side of a definition must not contain variables that are not already on the left hand side. In so far locale definitions behave like theory-level definitions. However, the locale concept realizes *dependent definitions* in that any variable that is fixed as a locale constant can occur on the right hand side of definitions. For example, a definition like

```
e_def "e == (G.<e>)"
```

contains the locale constant `G` on the right hand side. In principle, `G` is a free variable. Hence, this is a dependent definition. In Isabelle `defs` this would cause an error message \extra variable on right hand side". Naturally, definitions can already use the syntax of the locale constants in the `fixes` subsection. The `defines` part is, as the `assumes` part, optional.

Note also, that there are two different ways a locale constant can be used: one is to state its properties abstractly using rules, and one is to declare it as a name for a definition.

3.2 Invocation and Scope

After definition, locales may be opened and closed in a block-structured manner. The list (stack) of currently active locales is called *scope*. The operation for activating locales is *open*, the reverse one is *close*.

Scope The locale scope is part of each theory. It is a dynamic stack containing all active locales at a certain point in an interactive Isabelle session. The scope lives until all locales are explicitly closed. At any time there can be more than one locale open. The contents of these various active locales are all visible in the scope. Locales can be built by extension from other locales (cf. Sect. 3.3), i.e. they are nested. If a locale built by extension is open, the nesting is reflected in the scope, which contains the nested locales as layers. To check the state of the scope during a development the function `Print_scope` may be used. It displays the names of all open locales on the scope. The function `print_locales` applied to a theory displays all locales contained in that theory and in addition also the current scope.

Opening Locales can be *opened* at any point during an Isabelle session where we want to prove theorems concerning the locale. Opening a locale means making its contents visible by pushing it onto the scope of the current theory. Inside a scope of opened locales, theorems can use all definitions and rules contained in the locales on the scope. The rules and definitions may be accessed individually using the function *thm*. This function is applied to the names assigned to locale rules and definitions as strings. The opening command is called `Open_locale` and takes the name of the locale to be opened as its argument. In case of nested locales the opening command has to respect the nested structure (cf. Sect. 3.3).

Closing *Closing* means to cancel the last opened locale, pushing it off the scope. Theorems proved during the life time of this locale will be disabled, unless they have been explicitly exported, as described below. However, when the same locale is opened again these theorems may be used again as well, provided that they were saved as theorems in the first place, using `qed` or ML assignment. The command `Close_locale` takes a locale name as a string and checks if this locale is actually the topmost locale on the scope. If this is the case, it removes this locale, otherwise it prints a warning message and does not change the scope.

Export of Theorems Export of theorems transports theorems out of the scope of locales. Locale rules that have been used in the proof of a theorem inside a locale are carried by the exported form of the theorem as its individual meta-assumptions. The locale constants are universally quantified variables in the exported theorems, hence such theorems can be instantiated individually. Definitions become unfolded; locale constants that were merely used on the left hand side of a definition vanish. Logically, exporting corresponds to a combined application of introduction rules for implication and universal quantification. Exporting forms a kind of *normalization* of theorems in a locale scope.

According to the possibility of nested locales there are two different forms of export. The first one is realized by the function `export` that exports theorems through all layers of opened locales of the scope. Hence, the application of export to a theorem yields a theorem of the global level, that is, the current theory context without any local assumptions or definitions.

The other export function `Export` transports theorems just one level up in the scope. When locales are nested we might want to export a theorem, but not to the global level of the current theory, i.e. not outside all locales in the nesting, instead just to the previous level, because that is where we need it as a lemma. If we are in a nesting of locales of depth n, an application of `Export` will transform a theorem to one of level $n - 1$, i.e. into one that is independent of the definitions and assumptions of the locale that was on level n, but still uses locale constants, definitions and rules of the $n - 1$ locales underneath.

3.3 Other Aspects

Proofs The theorems proved inside a locale can use the locale rules as axioms, accessing them by their names. The used locale rules are held as meta-assumptions. Hence, subgoals created in a proof matching locale assumptions are solved automatically. Theorems proved in a locale can be exported as theorems of the global level under the assumption of the locale rules they use. If a theorem needs only a certain portion of the locale's assumptions, only those will be mentioned in the global form of the theorem.

Polymorphism Isabelle's meta-logic is based on a version of Church's Simple Theory of Types [Chu40] with schematic polymorphism. Free type variables are implicitly universally quantified at the outer level of declarations and statements. For example, a constant declaration

```
consts  f :: 'a => 'a
```

basically means that `f` has type $\forall \alpha .\, \alpha \Rightarrow \alpha$. So, if there is a subsequent constant declaration using the same type variable α, those are different type variables. That is, they can be instantiated *differently* in the same context.

Now, for locales the scope of polymorphic type variables is wider. The quantification of the type variables is placed at the outside of the locale. On the one hand, this difference allows us to define sharing of type domains of operators at an abstract level. This is important for the algebraic reasoning that we are focusing on in the examples. On the other hand, locale definitions may not be polymorphic within the locale's scope.

This feature solves the problem we encountered in case studies from abstract algebra, most prominently in the proof of Sylow's theorem [KP99]. Using earlier versions of Isabelle without locales, we had to choose a fixed type `i` in order to model the fixing of a polymorphic type of groups to enable readable formulas. Thereby, the final theorem was not applicable to arbitrary groups. In a more recent version of Sylow's theorem, using locales, we achieve the same syntax, and the result is generally applicable to all groups (cf. [Kam99a, Chapt. 6]).

Augmenting Locales As locales are defined statically in an Isabelle theory, operations on locales may be used to construct locales from other predefined ones statically in an Isabelle theory. A locale can be defined as the extension of a previously defined locale. This operation of extension is optional and is syntactically expressed as

```
locale foo = bar + ...
```

The locale `foo` builds on the constants and syntax of the locale `bar`. That is, all contents of the locale `bar` can be used in definitions and rules of the corresponding parts of the locale `foo`. Although locale `foo` assumes the `fixes` part of `bar` it does not automatically subsume its rules and definitions. Normally, one expects to use locale `foo` only if locale `bar` is already active. The opening mechanism is designed such that in the case of a locale built by extension it opens the ancestor automatically. If one opens a locale `foo` that is defined by extension from locale `bar` the function `Open_locale` checks if locale `bar` is open. If so, then it just opens `foo`, if not, then it prints a message and opens `bar` before opening `foo`. Naturally, this carries on, if `bar` is again an extension. The locales `bar` and `foo` become separate layers on the scope; `foo` has to be closed before `bar` can be closed (cf. Sect. 3.2).

In case of name clashes always the innermost definition is visible. That is, a name defined in a locale hides an equal name of a theory during the life time of the locale. When locales are built by extension, names may be hidden similarly. This is not possible if unrelated locales are opened simultaneously.

Another interesting device (which has not yet been implemented) is renaming of locale constants. This can be very useful if we want to have more than one instance of the same locale in the scope, for example when we reason with two different groups. The following illustrates a possible renaming mechanism: `loc_r` is created from `loc_c` by renaming all occurrences of locale constant `c` by `r`.

```
locale loc_r = loc_c [r/c]
```

Merging of locales by naming them could be another operation for locales. Although it seems similar to extension, one usually encounters difficulties because of shared ancestors.

4 Implementation Issues

In this section we briefly highlight some of the implementation issues of locales. In particular, we outline some key features of recent versions of Isabelle that enable to implement new theory definition features properly.

Extending the Isabelle theory language by any kind of new mechanism typically consists of the following stages:

(1) providing private theory data,
(2) writing a theory extension function,
(3) installing a new theory section parser.

For our particular mechanism of locales, we also have to adapt parts of the Isabelle goal package to cope with scopes as discussed in the previous section:

(4) modify term read and print functions,
(5) modify proof result operation.

4.1 Theory Data

Basically, any new theory extension mechanism boils down to already existing ones, like constant declarations and definitions. For example, the standard Isabelle/HOL `datatype` package could be seen just as a generator of huge amounts of types, constants, and theorems. This pure approach to theory extension has a severe drawback, though. It is like *compiling down* information, losing most of the original source level structure. E.g. it would be extremely hard to figure out any `datatype` specification (the set of constructors, say) from the soup of generated primitive extensions left behind in the theory.

The generic theory data concept, introduced in Isabelle98 and improved in later releases, offers a solution to this problem by enabling users to write packages in a *structure preserving* way. Thus one may declare named slots of *any* ML type to be stored within Isabelle theory objects. This way new extensions mechanisms may deposit appropriate source-level information as required later for any derived operation.

Picking up the `datatype` example again, there may be a generic induction tactic, that figures out the actual rule to apply from the type of some variable. This would be accomplished by doing a lookup in the private `datatype` theory data, containing full information about any HOL type represented as inductive datatype.

Note that traditionally in the LCF system approach, such data would be stored as values or structures within the ML runtime environment, with only very limited means to access this later from other ML programs. Breaking with this tradition, the recent Isabelle approach is more powerful, internalizing generic data as first-class components of theory objects.

The ML functor TheoryDataFun that is part of Isabelle/Pure provides a *fully type-safe* interface to generic data slots[1]. The argument structure is expected to have the following signature:

```
signature THEORY_DATA_ARGS =
sig
  val name: string
  type T
  val empty: T
  val merge: T * T -> T
  val print: Sign.sg -> T -> unit
end
```

Here name and T specify the new data slot by name and ML type, while empty gives its initial value. The merge operation is called when theories are joined, as should be the private data. Finally, print shall display the theory data in some human readable way; the function obtains the signature of the current theory (\self") as additional argument.

The result structure of TheoryDataFun is as follows:

```
signature THEORY_DATA =
sig
  type T
  val init: theory -> theory
  val print: theory -> unit
  val get: theory -> T
  val put: T -> theory -> theory
end
```

The new data slot has to be made known via above init operation. This is much like a run-time type declaration within a theory. Afterwards any derived theory knows about the print, get and put functions as given above.

For locales, we have defined a data slot called \Pure/locales" that contains a table of all defined locales, together with their hierarchical name space. There is also a reference variable of the current scope, containing a list of locales identifiers.

4.2 Theory Extension Function

Employing above private theory data slot, we have implemented the actual locale definition mechanism on top of usual Isabelle primitives (e.g. add_modesyntax). The ultimate result is the ML function add_locale, which is the actual theory extender that does all the hard work:

[1] This is achieved by invoking most of the black-magic that Standard ML has to offer: exception constructors for introducing new injections into type exn, private references as tags for identification and authorization, and functors for hiding. We see that ML is for the Real Programmer, after all!

```
  val add_locale: :::  -> theory -> theory
```

Here the dots refer to the locale specification, including `fixes`, `assumes`, `defines` arguments. After preparing these by parsing, type checking etc., we store the information via above `get` and `put` operations in our theory data slot, updating the table of existing locales. We also invoke a few other Isabelle primitives to extend the theory's syntax, for example.

4.3 Theory Section Parser

Another part of the scheme of adding a theory section to Isabelle is to provide a parsing method. The actual parser `locale_decl` for the locale definitions is just one ML-term constructed from parser combinators as are well-known in the functional programming community. Using Isabelle's `ThySyn.add_syntax` operation we can now associate our function `add_locale` with the `locale_decl` parser and plug it into the main theory syntax.

4.4 Interface

Apart from the actual theory extension function discussed above, there are a few more things to be done for the locale implementation.

The read and print functions of terms have to be adjusted to locales: if a locale is open, we want any term that is read in, to respect the bindings of types and terms of that locale. We augment the basic function `read_term` such that it checks if a locale is open, i.e. if the current scope is nonempty, and then bases the type inference on this information. Similarly, we adjust the function `pretty_term`. It is used to print proof states. Isabelle's goal package has been modified to use these read and print functions.

5 Examples from Abstract Algebra

We illustrate the use of the implementation by examples with the abstract algebraic structure of groups. We use a representation of groups that we found to be the most appropriate for abstract algebraic structures [Kam99b]. The base theory is `Group`. It contains the theory for groups. We define a basic pattern type for the simple structure of groups, by an extensible record definition [NW98][2].

```
  record 'a grouptype =
    carrier  :: "'a set"         ("_ .<cr>"  [10] 10)
    bin_op   :: "['a, 'a] => 'a" ("_ .<f>"  [10] 10)
    inverse  :: "'a => 'a"       ("_ .<inv>"  [10] 10)
    unit     :: "'a"             ("_ .<e>"  [10] 10)
```

[2] We use pretty printing facilities for records that are not yet available. The example remains the same, because one can achieve the same syntax using separate `syntax` declarations manually.

Now, we have defined a record type with four fields that gives us the projection functions to refer to the constituents of an element of this type. The class of all groups is defined as a typed HOL set over this record type [Kam99b]. This definition entails all the properties of a group and enables to state the group property quite concisely as

```
G : Group
```

Given that the Isabelle theory for groups contains the locale displayed in Sect. 3 we can now use it in an interactive Isabelle session. We open the locale `group` with the ML command

```
Open_locale "group";
```

Now the assumptions and definitions are visible, i.e. we are in the scope of the locale `group`. ML function `print_locales` shows all information about locales in the theory:

```
print_locales Group.thy;
```

This returns all information about the locale `group` and the current scope.[3]

```
locale name space:
  "Group.group" = "group", "Group.group"
locales:
  group =
    consts:
      G :: "'a set * (['a, 'a] => 'a) * ('a => 'a) * 'a"
      e :: "'a"
      binop :: "['a, 'a] => 'a"
      inv :: "'a => 'a"
    rules:
      Group_G: "G : Group"
    defs:
      e_def: "e == (G.<e>)"
      binop_def: "!!x y. binop x y == (G.<f>) x y"
      inv_def: "!!x. inv x == (G.<inv>) x"
current scope: group
```

Note, how the definitions with free variables have been bound by the meta-level universal quantifier (`!!`). The locale print function also gives information about the name spaces of the table of locales in the theory `Group` and displays the contents of the current scope.

As an illustration of the improvement we show how a proof for groups works now. Assuming that the theory of groups is loaded we demonstrate one proof that shows how the inverse can be swapped with the group operation.

```
Goal "[| x : (G.<cr>); y : (G.<cr>) |] ==> i(x # y) = (i y)#(i x)";
```

[3] The print function is mainly for inspecting and debugging, so the output of terms is in their actual internal form without locale syntax.

Isabelle sets the proof up and keeps the display of the dependent locale syntax.

```
1. !! x y. [| x : (G.<cr>); y : (G.<cr>) |] ==> i(x # y) = (i y)#(i x)
```

We can now perform the proof as usual, but with the nice abbreviations and syntax. We can apply all results which we might have proved about groups inside the locale. We can even use the syntax when we use tactics that use explicit instantiation, e.g. `res_inst_tac`. When the proof is finished, we can assign it to a name using `result()`. The theorem is now:

```
val inv_prod = "[| ?x : (G.<cr>); ?y : (G.<cr>) |]
  ==> inv (binop ?x ?y) = binop (inv ?y) (inv ?x)
  [!!x. inv x == (G.<inv>) x, G : Group,
   !!x y. binop x y == (G.<f>) x y, e == (G.<e>)]" : thm
```

As meta-assumptions annotated at the theorem we find all the used rules and definitions, the syntax uses the explicit names of the locale constants, not their pretty printing form. The question mark `?` in front of variables labels free schematic variables in Isabelle. They may be instantiated when applying the theorem. The assumption `e == (G.<e>)` is included because during the proof it was used to abbreviate the unit element.

To transform the theorem into its global form we just type `export inv_prod`.

```
"[| ?G : Group; ?x : (?G.<cr>); ?y : (?G.<cr>) |] ==>
(?G.<inv>)((?G.<f>) ?x ?y) = (?G.<f>)((?G.<inv>) ?y)((?G.<inv>) ?x)"
```

The locale constant `G` is now a free schematic variable of the theorem. Hence, the theorem is universally applicable to all groups. The locale definitions have been eliminated. The other locale constants, e.g. `binop`, are replaced by their explicit versions, and have thus vanished together with the locale definitions.

The locale facilities for groups are of course even more practical if we carry on to more complex structures like cosets. Assuming an adequate definition for cosets and products of subsets of a group (e.g. [Kam99b])

```
r_coset G H a == (  x. (G.<f>) x a) ‘‘ H
set_prod G H1 H2 == (  x. (G.<f>) (fst x)(snd x)) ‘‘ (H1 × H2)
```

where `‘‘` yields the image of a HOL function applied to a set | we use an extension of the locale for groups thereby enhancing the concrete syntax of the above definitions.

```
locale coset = group +
  fixes
    rcos      :: "['a set, 'a] => 'a set"     ("_ #> _" [60,61]60)
    setprod   :: "['a set, 'a set] => 'a set" ("_ <#> _" [60,61]60)
  defines
    rcos_def "H #> x == r_coset G H x"
    setprod_def "H1 <#> H2 == set_prod G H1 H2"
```

This enables us to reason in a natural way that reflects typical objectives of mathematics | in this case abstract algebra. We reason about the behaviour of substructures of a structure, like cosets of a group. Are they a group as well?[4] Therefore, we welcome a notation like

```
(H #> x) <#> (H #> y) = H #> (x # y)
```

when we have to reason with such substructural properties. While knowing that the underlying definitions are adequate and related properties derivable, we can reason with a convenient mathematical notation. Without locales the formula we had to deal with would be

```
set_prod G (r_coset G H x)(r_coset G H y) = r_coset G H ((G.<f>) x y)
```

The improvement is considerable and enhances comprehension of proofs and the actual finding of solutions | in particular, if we consider that we are confronted with the formulas not only in the main goal statements but in each step during the interactive proof.

6 Discussion

First of all, term syntax may be greatly improved by locales because they enable dependent local definitions. Locale constants can have pretty printing syntax assigned to them and this syntax can as well be dependent, i.e. use everything that is declared as fixed implicitly. So, locales approximate a natural mathematical style of formalization. Locales are a simpler concept than modules. They do not enable abstraction over type constructors. Neither do locales support polymorphic constants and definitions as the topmost theory level does.

On the other hand, these restrictions admit to define a representation of a locale as a *meta-logical predicate* fairly easily. Thereby, locales can be first-class citizen of the meta logic. We have developed this aspect of locales elsewhere [Kam99a]. In the latter experiment, we implemented the mechanical generation of a first-class representation for a locale. This implementation automatically extends the theory state of an Isabelle formalization by declarations and definitions for a predicate representing the locale logically. But, in many cases we do not think of a locale as a logical object, rather just an theory-level assembly of items. Then, we do not want this overhead of automatically created rules and constants. Consequently, we abandoned the automatic generation of a first class representation for locales. We prefer to perform the first-class reasoning separately in higher-order logic, using an approach with dependent sets [Kam99b].

In some sense, locales do have a first-class representation: globally interesting theorems that are proved in a locale may be exported. Then the former context structure of the locale gets dissolved: the definitions become expanded (and thus vanish). The locale constants turn into variables, and the assumptions become individual premises of the exported theorem. Although this individual representation of theorems does not entail the locale itself as a first-class citizen of the logic, the context structure of the locale is translated into the meta-logical

[4] They are a group if H is normal which is proved conveniently in Isabelle with locales.

structure of assumptions and theorems. In so far we mirror the local assumptions | that are really the locale | into a representation in terms of the simple structural language of Isabelle's meta-logic. This translation corresponds logically to an application of the introduction rules for implication and the universal quantifier of the meta-logic. And, because Isabelle has a proper meta-logic this first-class representation is easy to apply.

Generality of proofs is partly revealed in locales: certain premises that are available in a locale are not used at all in the proof of a theorem. In that case the exported version of the theorem will not contain these premises. This may seem a bit exotic, in that theorems proved in the same locale scope might have different premise lists. That is, theorems may generally just contain a subset of the locale assumptions in their premises. That takes away uniformity of theorems of a locale but grants that theorems may be proved in a locale and will be individually considered for the export. In many cases one discovers that a theorem that one closely linked with, say, groups actually does not at all depend on a specific group property and is more generally valid. That is, locales filter the theorems to be of the most general form according to the locale assumptions.

Locales are, as a concept, of general value for Isabelle independent of abstract algebraic proof. In particular, they are independent of any object logic. That is, they can be applied merely assuming the meta-logic of Isabelle/Pure. They are already applied in other places of the Isabelle theories, e.g. for reasoning about finite sets where the fixing of a function enhances the proof of properties of a \fold" functional and similarly in proofs about multisets and the formal method UNITY [CM88]. Furthermore, the concept can be transferred to all higher-order logic theorem provers. There are only a few things the concept relies on. In particular, the features needed are implication and universal quantification | the two constructors that build the basis for the reflection of locales *via* export and are at the same time the explanation of the meaning of locales. For theorem provers where the theory infrastructure differs greatly from Isabelle's, one may consider dynamic definition of locales instead of the static one.

The simple implementation of the locale idea as presented in this paper works well together with the first-class representation of structures by an embedding using dependent types [Kam99b]. Both concepts can be used simultaneously to provide an adequate support for reasoning in abstract algebra.

References

[CCF$^+$96] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, and C. Muñoz. *The Coq Proof Assistant User's Guide, version 6.1.* INRIA-Rocquencourt et CNRS-ENS Lyon, 1996.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, pages 56{68, 1940.

[CM88] K. Mani Chandi and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[dB80] N. G. de Bruijn. A Survey of the Project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic*, Academic Press Limited, pages 579{606. 1980.

[Dow90] G. Dowek. Naming and Scoping in a Mathematical Vernacular. Technical Report 1283, INRIA, Rocquencourt, 1990.

[FGT93] W. M. Farmer, J. D. Guttman, and F. J. Thayer. imps: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213{248, 1993.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andres Modet, and Jeannette M. Wing.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Kam99a] F. Kammüller. *Modular Reasoning in Isabelle*. PhD thesis, University of Cambridge, 1999. submitted.

[Kam99b] F. Kammüller. Modular Structures as Dependent Types in Isabelle. In *Types for Proofs and Programs: TYPES '98*, LNCS. Springer-Verlag, 1999. Selected papers. To appear.

[KP99] F. Kammüller and L. C. Paulson. A Formal Proof of Sylow's First Theorem { An Experiment in Abstract Algebra with Isabelle HOL. *Journal of Automated Reasoning*, 1999. To appear.

[NW98] W. Naraschewski and M. Wenzel. Object-oriented Verification based on Record Subtyping in Higher-Order Logic. In *11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, ANU, Canberra, Australia, 1998. Springer-Verlag.

[ORR$^+$96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[OSRSC98] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS Language Reference. Part of the PVS Manual. Available on the Web as `http://www.csl.sri.com/pvsweb/manuals.html`, September 1998.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Win93] P. J. Windley. Abstract Theories in HOL. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20, pages 197{210. North-Holland, 1993.

Isar – A Generic Interpretative Approach to Readable Formal Proof Documents

Markus Wenzel[*]

Technische Universität München
Institut für Informatik, Arcisstraße 21, 80290 München, Germany
http://www.in.tum.de/~wenzelm/

Abstract. We present a generic approach to readable formal proof documents, called *Intelligible semi-automated reasoning* (*Isar*). It addresses the major problem of existing interactive theorem proving systems that there is no appropriate notion of proof available that is suitable for human communication, or even just maintenance. Isar's main aspect is its formal language for natural deduction proofs, which sets out to bridge the semantic gap between internal notions of proof given by state-of-the-art interactive theorem proving systems and an appropriate level of abstraction for user-level work. The Isar language is both human readable and machine-checkable, by virtue of the Isar/VM interpreter.
Compared to existing declarative theorem proving systems, Isar avoids several shortcomings: it is based on a few basic principles only, it is quite independent of the underlying logic, and supports a broad range of automated proof methods. Interactive proof development is supported as well. Most of the Isar concepts have already been implemented within Isabelle. The resulting system already accommodates simple applications.

1 Introduction

Interactive theorem proving systems such as HOL [10], Coq [7], PVS [15], and Isabelle [16], have reached a reasonable level of maturity in recent years. On the one hand supporting expressive logics like set theory or type theory, on the other hand having acquired decent automated proof support, such systems provide quite powerful environments for sizeable applications. Taking Isabelle/HOL as an arbitrary representative of these *semi-automated reasoning* systems, typical applications are the formalization of substantial parts of the Java type system and operational semantics [14], formalization of the first 100 pages of a semantics textbook [13], or formal proof of Church-Rosser property of β-reductions [12].

Despite this success in actually formalizing parts of mathematics and computer science, there are still obstacles in addressing a broad range of users. One of the main problems is that, paradoxically, none of the major semi-automated reasoning systems support an adequate *primary* notion of proof that is amenable to human understanding. Typical prover input languages are rather arcane, demanding a steep learning curve of users to write any proof scripts at all. Even

[*] Research supported by DFG grant Ni 491/2-1.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 167–183, 1999.

worse, the resulting texts are very difficult to understand, usually requiring stepwise replay in the system to make anything out of it. This situation is bad enough for proof maintenance, but is impossible for communicating formal proofs | the fruits of the formalization effort | to a wider audience.

According to folklore, performing proof is similar to programming. Comparing current formal proof technology with that of programming languages and methodologies, though, we seem to be stuck at the assembly language level. There are many attempts to solve this problem, like providing user interfaces for theorem provers that help users to put together proof scripts, or browser tools presenting the prover's internal structures, or generators and translators to convert between different notions of proof | even natural language.

The Mizar System [19, 22] pioneered a different approach, taking the issue of a human-readably *proof language* seriously. More recently, several efforts have been undertaken to transfer ideas of Mizar into the established tradition of tactical theorem proving, while trying to avoid its well-known shortcomings. The DECLARE system [20, 21] has been probably the most elaborate, so far.

Our approach, which is called *Intelligible semi-automated reasoning* (*Isar*), can be best understood in that tradition, too. We aim to address the problem at its very core, namely the primary notion of formal proof that the systems offers to its users, authors and audience alike. Just as the programming community did some decades ago, we set out to develop a high-level formal language for proofs that is designed with the human in mind, rather than the machine.

The Isar language framework, taking both the underlying logic and a set of proof methods as parameters, results in an environment for \declarative" natural deduction proofs that may be \executed" at the same time. Checking is achieved by the *Isar virtual machine* interpreter, which also provides an operational semantics of the Isar proof language.

Thus the Isar approach to readable formal proof documents is best characterized as being *interpretative*. It offers a higher conceptual level of formal proof that shall be considered as the new *primary* one. Any internal inferences taking place within the underlying deductive system are encapsulated in an abstract judgment of derivability. Any function mapping a proof goal to an appropriate proof rule may be incorporated as *proof method*. Thus arbitrary automated proof procedures may be integrated as opaque refinement steps.

This immediately raises the question of soundness, which is handled in Isar according to the *back-pressure* principle. The Isar/VM interpreter refers to actual inferences at the level below only *abstractly*, without breaching its integrity. Thus we inherit whatever notion of correctness is available in the underlying inference system (e.g. primitive proof terms). Basically, this is just the well-known \LCF approach" of correctness-by-construction applied at the level of the Isar/VM.

The issue of *presentation* of Isar documents can be kept rather trivial, because the proof language has been designed with readability already in mind. Some pretty printing and pruning of a few details should be sufficient for reasonable output. Nevertheless, Isar could be easily put into a broader context of more advanced presentation concepts, including natural language generation (e.g. [8]).

The rest of this paper is structured as follows. Section 2 presents some example proof documents written in the Isar language. Several aspects of Isar are discussed informally as we proceed. Section 3 reviews formal preliminaries required for the subsequent treatment of the Isar language. Section 4 introduces the Isar formal proof language syntax and operational semantics, proof methods, and extra-logical features.

2 Example Isar Proof Documents

Isar provides a generic framework for readable formal proofs that supports a broad range of both logics and proof tools. A typical instantiation for actual applications would use Higher-order Logic [6] together with a reasonable degree of automation [17, 18]. Yet the main objective of Isar is not to achieve shorter proofs with more automation, but better ones. The writer is enabled to express the interesting parts of the reasoning as explicit proof text, while leaving other parts to the machine. For the sake of the following examples, which are from pure first-order logic (both intuitionistic and classical), we refer to very simple proof methods only.

2.1 Basic Proofs

To get a first idea what natural deduction proofs may look like in Isar, we review three well-known propositions from intuitionistic logic: $I{:}\,A\ !\ A$ and $K{:}\,A\ !\ B\ !\ A$ and $S{:}\,(A\ !\ B\ !\ C)\ !\ (A\ !\ B)\ !\ A\ !\ C$; recall that $!$ is nested to the right.

```
theorem I: A ! A
proof
  assume A
  show A :
qed
```

Unsurprisingly, the proof of I is rather trivial: we just assume A in order to show A again. The dot \:" denotes an *immediate proof*, meaning that the current problem holds by assumption.

```
theorem K: A ! B ! A
proof
  assume A
  show B ! A
  proof
    show A :
  qed
qed
```

Only slightly less trivial is K: we assume A in order to show $B\ !\ A$, which holds because we can show A trivially. Note how proofs may be nested at any time, simply by stating a new problem (here via **show**). The subsequent proof (delimited by a **proof**/**qed** pair), is implicitly enclosed by a *logical block* that inherits the current context (assumptions etc.), but keeps any changes local. Block structure, which is a well-known principle from programming languages, is an important starting point to achieve structured proofs (e.g. [9]). Also note that there are implicit default proof methods invoked at the beginning (**proof**) and end (**qed**) of any subproof. The initial method associated with **proof** just picks standard introduction and elimination rules automatically according to the topmost symbol involved (here ! introduction), the terminal method associated with **qed** solves all remaining goals by assumption. Proof methods may be also specified explicitly, as in **proof** (*rule modus-ponens*)".

```
theorem S: (A ! B ! C) ! (A ! B) ! A ! C
proof
  assume A ! B ! C
  show (A ! B) ! A ! C
  proof
    assume A ! B
    show A ! C
    proof
      assume A
      show C
      proof (rule modus-ponens)
        show B ! C by (rule modus-ponens)
        show B by (rule modus-ponens)
      qed
    qed
  qed
qed
```

In order to prove S we first decompose the three topmost implications, represented by the **assume**/**show** pairs. Then we put things together again, by applying *modus ponens* to get C from B ! C and B, which are themselves established by *modus ponens* (**by** abbreviates a single-step proof in terminal position). Note that the context of assumptions A ! B ! C, A ! B, A is taken into account implicitly where appropriate.

What have we achieved so far? Certainly, there are more compact ways to write down natural deduction proofs. In typed -calculus our examples would read $x{:}A{:}\ x$ and $x{:}A\ y{:}B{:}\ x$ and $x{:}A$! B ! $C\ y{:}A$! $B\ z{:}A{:}\ (x\,z)\,(y\,z)$.

The Isar text is much more verbose: apart from providing fancy keywords for arranging the proof, it explicitly says at every stage which statement is established next. Speaking in terms of -calculus, we have given types to actual subterms, rather than variables only. This sort of redundancy has already been observed in the ProveEasy teaching tool [5] as very important ingredient to improve readability of formal proofs. Yet one has to be cautious not to become too verbose, lest the structure of the reasoning be obscured again. Isar already leaves some inferences implicit, e.g. the way assumptions are applied. Moreover, the level of primitive inferences may be transcended by appealing to automated proof procedures, which are treated as opaque refinement steps (cf. *x*2.4).

2.2 Mixing Backward and Forward Reasoning

The previous examples have been strictly backward. While any proof may in principle be written this way, it may not be most natural. Forward style is often more adequate when working from intermediate facts.

Isar offers both backward and forward reasoning elements, as an example consider the following three proofs of *A* ^ *B* ! *B* ^ *A*.

```
lemma A ^ B ! B ^ A
proof
  assume A ^ B
  show B ^ A
  proof
    show B
      by (rule conj_2)
    show A
      by (rule conj_1)
  qed
qed
```

```
lemma A ^ B ! B ^ A
proof
  assume A ^ B
  then
    show B ^ A
  proof
    assume A; B
    show ??thesis ::
  qed
qed
```

```
lemma A ^ B ! B ^ A
proof
  assume ab: A ^ B
  from ab
    have a: A ::
  from ab
    have b: B ::
  from b; a
    show B ^ A ::
qed
```

The first version is strictly backward, just as the examples of *x*2.1. We have to provide the projections $conj_{1;2}$ explicitly, because the corresponding goals do not provide enough syntactic structure to determine the next step. This may be seen as an indication that forward reasoning would be more appropriate.

Consequently, the second version proceeds by forward chaining from the assumption *A* ^ *B*, as indicated by then. This corresponds to ^ elimination, i.e. we may assume the conjuncts in order to show again *B* ^ *A*. Repeating the current goal is typical for elimination proofs, so Isar provides a way to refer to it symbolically as ??*thesis*. The double dot *::*" denotes a *trivial proof*, by a single standard rule. Alternatively, we could have written \by (*rule conj-intro*)".

Forward chaining may be done not only from the most recent fact, but from any one available in the current scope. This typically involves naming intermediate results (assumptions, or auxiliary results introduced via have) and referring to them explicitly via from. Thus the third proof above achieves an extreme case of forward-style reasoning, with only the outermost step being backward.

The key observation from these examples is that there is more to readable natural deduction than pure -calculus style reasoning. Isar's then language element can be understood as *reverse* application of -terms. Thus elimination proofs and other kinds of forward reasoning are supported as first-class concepts.

Leaving the writer the choice of proof direction is very important to achieve readable proofs, although yielding a nice balance between the extremes of purely forward and purely backward requires some degree of discernment. As a rule of thumb for good style, backward steps should be the big ones (decomposition, case analysis, induction etc.), while forward steps typically pick up assumptions or other facts to achieve the next result in a few small steps.

As a more realistic example for mixed backward and forward reasoning consider Peirce's law, which is a classical theorem so its proof is by contradiction. Backward-only proof would be rather nasty, due to the ! -nesting.

```
theorem Peirce′s-Law: ((A ! B) ! A) ! A
proof                                                                  [:A]
  assume ab-a: (A ! B) ! A                                               ⋮
  show A                                                                 A
  proof (rule contradiction)     −− use classical contradiction rule:  ───
                                                                         A
    assume not-a: :A

    have ab: A ! B
    proof
      assume a: A
      from not-a; a show B ::
    qed

    from ab-a; ab show A ::
  qed
qed
```

There are many more ways to arrange the reasoning. In the following variant we swap two sub-proofs of the contradiction. The result looks as if a *cut* had been performed. (The *d*{*c* parentheses are a version of begin{end.)

```
theorem Peirce′s-Law: ((A ! B) ! A) ! A
proof
  assume ab-a: (A ! B) ! A
  show A
  proof (rule contradiction)
 d  assume ab: A ! B              −− to be proved later (   cut)
    from ab-a; ab show A :: c

    assume not-a: :A
    show A ! B
    proof
      assume a: A
      from not-a; a show B ::
    qed
  qed
qed
```

Which of the two variants is actually more readable is a highly subjective question, of course. The most appropriate arrangement of reasoning steps also depends on what the writer wants to point out to the audience in some particular situation. Isar does not try to enforce any particular way of proceeding, but aims at offering a high degree of flexibility.

2.3 Intra-logical and Extra-logical Binding of Variables

Leaving propositional logic behind, we consider $(9x\colon P(f(x))) \; ! \; (9x\colon P(x))$. Informally, this holds since after assuming $9x\colon P(f(x))$, we may fix some a such that $P(f(a))$ holds, and use $f(a)$ as witness for x in $9x\colon P(x)$ (note that the two bound variables x are in separate scopes). So the proof is just a composition of ! introduction, 9 elimination, 9 introduction. Writing down a natural deduction proof tree would result in a very compact and hard to understand representation of the reasoning involved, though. The Isar proof below tries to mimic our informal explanation, exhibiting many (redundant) details.

```
lemma (9x: P(f(x))) ! (9x: P(x))
proof
  assume 9x: P(f(x))
  then show 9x: P(x)
  proof (rule ex-elim)       -- use 9 elimination rule:
    fix a
    assume P(f(a)) (is P(??witness))
    show ??thesis by (rule ex-intro [with P ??witness])
  qed
qed
```

$$\frac{9x\colon A(x) \qquad \begin{matrix}[A(x)]_x \\ \vdots \\ B\end{matrix}}{B}$$

After forward chaining from fact $9x\colon P(f(x))$, we have locally fixed an arbitrary a (via fix) and assumed that $P(f(a))$ holds. In order to stress the rôle of the constituents of this statement, we also say that $P(f(a))$ matches the pattern $P(\text{??}witness)$ (via is). Equipped with all these parts, the thesis is finally established using 9 introduction instantiated with P and the ??*witness* term.

Above example exhibits two different kinds of variable binding. First fix a, which introduces a local Skolem constant used to establish a quantified proposition as usual. Second (is $P(\text{??}witness)$), which defines a local abbreviation for some term by higher-order matching, namely ??*witness* $f(a)$. The subsequent reasoning refers to a from *within* the logic, while abbreviations have a quite different logical status: being expanded before actual reasoning, the underlying logic engine will never see them. In a sense, this just provides an extra-logical illusion, yet a very powerful one.

Term abbreviations are also an important contribution to keep the Isar language lean and generic, avoiding separate language features for logic-specific proof idioms. Using appropriate proof methods together with abbreviations having telling names like ??*lhs*, ??*rhs*, ??*case* already provides sufficient means for representing typical proofs by calculation, case analysis, induction etc. nicely. Also note that ??*thesis* is just a special abbreviation that happens to be bound automatically | just consider any new goal goal implicitly decorated by (is ??*thesis*). Note that \thesis" is a separate language element in Mizar [22].

2.4 Automated Proof Methods and Abstraction

The quantifier proof of x2.3 has been rather verbose, intentionally. We have chosen to provide proof rules explicitly, even given instantiations. As it happens, these rules and instantiations can be figured out by the basic mechanism of picking standard introduction and elimination rules that we have assumed as the standard initial proof method so far.

```
lemma (9x: P(f(x))) ! (9x: P(x))
proof
  assume 9x: P(f(x))
  then show 9x: P(x)
  proof
      fix a
    assume P(f(a))
    show ??thesis ::
qed
```

Much more powerful automated deduction tools have been developed over the last decades, of course. From the Isar perspective, any of these may be plugged into the generic language framework as particular proof methods. Thus we may achieve more abstract proofs beyond the level of primitive rules, by letting the system solve open branches of proofs automatically, provided the situation has become sufficiently \obvious". In the following version of our example we have collapsed the problem completely by a single application of method *blast*", which shall refer to the generic tableau prover tactic integrated in Isabelle [18].

```
lemma (9x: P(f(x))) ! (9x: P(x)) by (blast)
```

Abstraction via automation gives the writer an additional dimension of choice in arranging proofs, yet a limited one, depending on the power of the automated tools available. Thus we achieve *accidental abstraction* only, in the sense that more succinct versions of the proof text still happen to work.

In practice, there will be often a conflict between the level of detail that the writer wishes to confer to his audience, and the automatic capabilities of the system. Isar also provides a simple mechanism for *explicit abstraction*. Subproofs started by proof$^?$/by$^?$ rather than proof/by are considered below the current *level of interest* for the intended audience. Thus excess detail may be easily pruned by the presentation component, e.g. printed as ellipsis (\. . . ").

In a sense, proof$^?$/by$^?$ have the effect of turning concrete text into an ad-hoc proof method (which are always considered opaque in Isar). More general means to describe methods would include parameters and recursion. This is beyond the scope of Isar, though, which is an environment for writing actual proofs rather than proof methods. Isar is left computationally incomplete by full intention. High-level languages for proof methods are an issue in their own right [1].

3 Formal Preliminaries

3.1 Basic Mathematical Notations

Functions. We write function application as $f\ x$ and use -abstraction $x\colon f(x)$. Point-wise update of functions is written postfix, $f[x_1 := \colon\colon\colon := x_n := y]$ denotes the function mapping $x_1;\colon\colon\colon;x_n$ to y and any other x to $f(x)$. Sequential composition of functions f and g (from left to right) is written $f;g$ which is defined as $(f;g)(x) = g\ (f\ x)$. Any of these operations may be used both for total functions ($A \to B$) and partial functions ($A \rightharpoonup B$).

Records are like tuples with explicitly labeled fields. For any record $r\colon R$ with some field $a\colon A$ the following operations are assumed: selector *get-a*$\colon R \to A$, update *set-a*$\colon A \to (R \to R)$, and the functional *map-a*$\colon (A \to A) \to (R \to R)$ which is defined as *map-a* $f \equiv \lambda r.$ *set-a* $(f$ (*get-a* $r))$.

Lists. Let *list of* A be the set of lists over A. We write $[x_1, \ldots, x_n]$ for the list of elements $x_1, \ldots, x_n$. List operations include x # *xs* (cons) and *xs* @ *ys* (append).

3.2 Lambda-Calculus and Natural Deduction

Most of the following concepts are from λ-Prolog or Isabelle [16, Part I].

λ*-Terms* are formed as (typed) constants or variables (from set *var*), by application $t\ u$, or abstraction $\lambda x.\ t$. We also take α-, β-conversions for granted.

Higher-order Abstract Syntax. Simply-typed λ-terms provide a means to describe abstract syntax adequately. Syntactic entities are represented as types, and constructors as (higher-order) constants. Thus tree structure is achieved by (nested) application, variable binding by abstraction, and substitution by β-reduction.

Natural Deduction (Meta-logic). We consider a minimal $\Rightarrow/\forall$-fragment of intuitionistic logic. For the abstract syntax, fix type *prop* (meta-level propositions), and constants $\Rightarrow\colon$ *prop* $\to$ *prop* $\to$ *prop* (nested to the right), $\forall\colon (\alpha \to$ *prop*$) \to$ *prop*. We write $[\varphi_1, \ldots, \varphi_n] \Rightarrow \varphi$ for $\varphi_1 \Rightarrow \ldots \Rightarrow \varphi_n \Rightarrow \varphi$, and $\forall x.\ P\ x$ for $\forall(\lambda x.\ P\ x)$. Deduction is expressed as an inductive relation $\Gamma \vdash \varphi$, by the usual rules of assumption, and $\Rightarrow/\forall$ introduction and elimination. Note that the corresponding proof trees can be again seen as λ-terms, although at a different level, where propositions are types. The set “$\vdash$” is also called “*theorem*”.

Encoding Object-logics. A broad range of natural deduction logics may now be encoded as follows. Fix types i of individuals, o of formulas, and a constant $D\colon o \to$ *prop* (for expressing derivability). Object-level natural deduction rules are represented as meta-level propositions, e.g. $\exists\colon (i \to o) \to o$ elimination as $D(\exists x.\ P\ x) \Rightarrow (\forall x.\ D(P\ x) \Rightarrow D(Q)) \Rightarrow D(Q)$. Let *form* be the set of propositions $D(A)$, for $A\colon o$. D is usually suppressed and left implicit. Object-level rules typically have the form of nested meta-level horn-clauses.

4 The Isar Proof Language

The Isar framework takes a meta-logical formulation (see §3.2) of the underlying logic as parameter. Thus we abstract over logical syntax, rules and automated proof procedures, which are represented as functions yielding rules (see §4.2).

4.1 Syntax

For the subsequent presentation of the Isar core syntax, *var* and *form* are from the underlying higher-order abstract syntax (see X3.2), *name* is any infinite set, while the *method* parameter refers to proof methods (see X4.2).

```
theory-stmt = theorem [name:] form proof
            j lemma [name:] form proof
            j types ::: j consts ::: j defs ::: j :::
      proof = proof [(method)] stmt  qed [(method)]
       stmt = begin stmt  end
            j note name = name+
            j fix var+
            j assume [name:] form+
            j then goal-stmt
            j goal-stmt
  goal-stmt = have [name:] form proof
            j show [name:] form proof
```

The actual Isar proof language (*proof*) is enclosed into a theory specification language (*theory-stmt*) that provides global statements theorem or lemma, entering into *proof* immediately, but also declarations and definitions of any kind. Note that advanced definitional mechanisms may also require proof. Enclosed by proof/qed, optionally with explicit initial or terminal method invocation, *proof* mainly consists of a list of local statements (*stmt*). This marginal syntactic rôle of *method* is in strong contrast to existing tactical proof languages.

Optional language elements above default to \it:" for result names, proof method specifications \(*single*)" for proof, \(*assumption*)" for qed (cf. X4.3).

A few well-formedness conditions of Isar texts are not yet covered by the above grammar. Considering begin{end and show/have{qed as block delimiters, we require any *name* reference to be well-defined in the current scope. Furthermore, the paths of variables introduced by fix may not contain duplicates, and then may only occur directly after note, assume, or qed.

Next the Isar core language is extended by a few derived elements. (Below *same* and *single* refer to standard proof methods introduced in X4.3.)

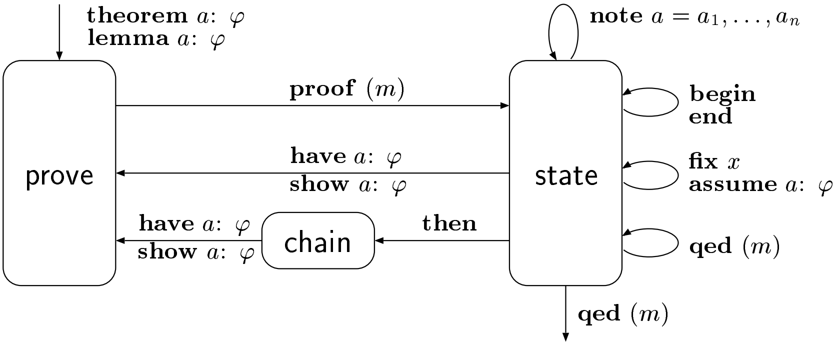
from $a_1; ::: ; a_n$	note facts = $a_1; ::: ; a_n$ then	
hence	then have	
thus	then show	
by (m)	proof (m) qed	(\terminal proof")
::	proof (*single*) qed	(\trivial proof")
:	proof (*same*) qed	(\immediate proof")

Basically, this is already the full syntax of the Isar language framework. Any logic-specific extensions will be by abbreviations or proof methods only.

4.2 Operational Semantics

The canonical operational semantics of Isar is given by direct interpretation within the *Isar virtual machine* (*Isar/VM*). Well-formed proof texts, which have tree structure if considered as abstract syntax entities, are translated into lists of Isar/VM instructions in a rather trivial way (the translation is particularly easy, because Isar lacks recursion): any syntactic construct, as indicated by some major keyword (proof, begin, show, etc.), simply becomes a separate instruction, which acts as transformer of the machine configuration.

Before going into further details, we consider the following abstract presentation of the Isar/VM. The machine configuration has three modes of operation, depicted as separate states prove, state, chain below. Transitions are marked by the corresponding Isar/VM instructions.



Any legal path of Isar/VM transitions constitutes an (interactive) proof, starting with theorem/lemma and ending eventually by a final qed. Intermediately, the two main modes alternate: prove (read \apply some method to refine the current problem") and state (read \build up a suitable environment to produce the next result"). Minor mode chain modifies the next goal accordingly.

More precisely, the Isar/VM configuration is a non-empty list of levels (for block structure), where each level consists of a record with the following fields:

mode : prove j state j chain
fixes : *list of var*
asms : *list of form*
results : *name* * *list of theorem*
problem : (*bool* *name*) ((*list of theorem*) *theorem*) j none

Fields *fixes* and *asms* constitute the Skolem and assumption context. The *results* environment collects lists of intermediate theorems, including the special one \facts" holding the most recent result (used for forward chaining). An open *problem* consists of a flag (indicating if the finished result is to be used to refine an enclosing goal), the result's name, a list of facts for forward chaining, and the actual goal (for *theorem* see x3.2). Goals are represented as rules according to

Isabelle tradition [16, Part I]: Γ ‘ $[(\ _1\ )\ '_1);:::;(\ _n\ )\ '_n)]\ )$ means that in assumption context Γ, main goal is to be shown from n subgoals $_i\ )\ '_i$.

An initial configuration, entered by **theorem** a: ’ or **lemma** a: ’, has a single level with fields *mode* = prove, *fixes* = [], *asms* = [], *results* = f(facts; [])g, and *problem* = ((false; a); ([]; ‘ ’) ’)). The terminal configuration is []. Intermediate transitions are by the (partial) function $T[\![I]\!]$, mapping Isar/VM configurations depending on instruction I and the current *mode* value as follows. Basic record operations applied to a configuration refer to the topmost level.

```
mode = prove :
  T[[proof (m)]]   refine-problem m; set-mode state
mode = state :
  T[[note a = a_1; ::: ; a_n]]
      map-results (  r: r [facts := a := r(a_1) @    @ r(a_n)])
  T[[begin]]   reset-facts; open-block; set-problem none
  T[[end]]   close-block
  T[[fix x]]   map-fixes (  xs: xs @ [x]); reset-facts
  T[[assume a:  '_1; ::: ; '_n]]
      map-asms (   :   @ [ '_1; ::: '_n]);
      map-results (  r: r [facts := a := [ '_1  ‘  '_1; ::: ; '_n  ‘  '_n]])
  T[[qed (m)]]   refine-problem m; check-result; apply-result; close-block
  T[[then]]   set-mode chain
  T[[have a:  ']]   setup-problem (false; a) (false;  ')
  T[[show a:  ']]   setup-problem (true; a) (false;  ')
mode = chain :
  T[[have a:  ']]   setup-problem (false; a) (true;  ')
  T[[show a:  ']]   setup-problem (true; a) (true;  ')

open-block (x   xs)    x   x   xs
close-block (x   xs)    xs
reset-facts    map-results (  r: r [facts := []])
setup-problem result-info (use-facts;  ') c    f c   where
        get-asms c
  facts    if use-facts then (get-results c facts) else []
  f    reset-facts; open-block; set-mode prove;
      set-problem (result-info; (facts;    ‘ (    )   ') )   '))
refine-problem m
  map-problem (  (x; (y; goal)): (x; (y; backchain goal (m facts)))
check-result c    c   if problem has no subgoals, else undefined
apply-result (x   xs)    x   (f xs)   where
  ((use-result; a); (y; res))    get-problem x
  result
    generalise (get-fixes x) (discharge (get-asms x − get-asms xs) res)
  f    map-results (  r: r [facts := a := [result]]);
      if use-result then refine-problem (  y: result) else (  x: x)
```

Above operation *setup-problem* initializes a new problem according the current assumption context and forward chaining mode etc. The goal is set up to establish result ’ from ’ under the assumptions.

Operation *refine-problem* back-chains (using a form of meta-level *modus ponens*) with the method applied to the facts to be used for forward chaining.

Operation *apply-result* modifies the upper configuration, binding the result and refining the second topmost problem wrt. block structure (if *use-result* had been set). Note that *generalise* and *discharge* are basic meta-level rules.

We claim (a proof is beyond the scope of this paper) that the Isar/VM is correct and complete as follows. For any ’, there is a path of transitions from the initial to the terminal configuration *i* ‘ ’ is a theorem derivable by natural deduction, assuming any rule in the image of the set of methods.

This result would guarantee that the Isar/VM does not fail unexpectedly, or produce unexpected theorems. Actual correctness in terms of formal derivability is achieved differently, though. Applying the well-known \LCF approach" of correctness-by-construction at the level of the Isar/VM implementation, results are always actual theorems, relative to the primitive inferences underlying proof methods and bookkeeping operations such as *refine-problem*.

4.3 Standard Proof Methods

In order to turn the Isar/VM into an actually working theorem proving system, some standard proof methods have to be provided. We have already referred to a few basic methods like *same*, *single*, which are defined below.

We have modeled proof methods as functions producing appropriate rules (meta-level theorems), which will be used to refine goals by backchaining. Operationally, this corresponds to a function reducing goal Γ ‘ $)$ to Γ ‘ ^{0} $)$ (leaving the hypotheses and main goal unchanged). This coincides with tactic application, only that proof methods may depend on facts for forward chaining. For the subsequent presentation we prefer to describe methods according to this operational view.

Method *same*" inserts the facts to any subgoal, which are left unchanged otherwise; *rule a*" applies rule a by back-chaining, after forward-chaining from facts; *single*" is similar to *rule*", but determines a standard introduction or elimination rule from the topmost symbol of the goal or the first fact automatically; *assumption*" solves subgoals by assumption.

These methods are sufficient to support primitive proofs as presented in x2. A more realistic environment would provide a few more advanced methods, in particular automated proof tools such as a generic tableau prover *blast*" [18], a higher-order simplifier *simp*" etc. A sufficiently powerful combination of such proof tools could be even made the default for **qed**. In contrast, the initial **proof** method should not be made too advanced by default, lest the subsequent proof text be obscured by the left-over state of its invocation. In DECLARE [21] automated *initial* proof methods are rejected altogether because of this.

4.4 Extra-logical Features

Isar is not a monolithic all-in-one language, but a hierarchy of concepts having different logical status. Apart from the core language considered so far, there are additional extra-logical features without semantics at the logical level.

Term Abbreviations. Any goal statement (show etc.) may be annotated with a list of term abbreviation patterns (is pat_1 : : : is pat_n). Alternatively, abbreviations may be bound by explicit let *pat* *term* statements.

Levels of Interest decorate proof/by commands by a natural number or ? (for infinity), indicating that the subsequent proof block becomes less interesting for the intended audience. The presentation component will use these hints to prune excess detail, collapsing it to ellipsis (\. . . "), for example.

Formal Comments of the form \−− *text*" may be associated with any Isar language element. The comment *text* may contain any text, which may again contain references to formal entities (terms, formulas, theorems etc.).

5 Conclusion and Related Work

We have proposed the generic *Intelligible semi-automated reasoning* (*Isar*) approach to readable formal proof documents. Its main aspect, the Isar formal proof language, supports both \declarative" proof texts and machine-checking, by direct \execution" within the Isar virtual machine (Isar/VM). While Isar does not require automation for basic operation, arbitrary automated deduction tools may be included in Isar proofs as appropriate. Automated tools are an important factor in scalability for realistic applications, of course.

Isar is most closely related to \declarative" theorem proving systems, notably Mizar [19, 22]. The Mizar project, started in the 1970s, has pioneered the idea of performing mathematical proof in a structured formal language, while hiding operational detail as much as possible. The gap towards the underlying calculus level is closed by a specific notion of *obvious inferences*. Internally, Mizar does not actually reduce its proof checking to basic inferences. Thus Mizar proofs are occasionally said to be \rigorous" only, rather than \formal".

Over the years, Mizar users have built up a large body of formalized mathematics. Despite this success, though, there are a few inherent shortcomings preventing further scalability for large applications. Mizar is not generic, but tightly built around its version of typed set theory and the *obvious inferences* prove checker. Its overall design has become rather baroque, such that even re-engineering the syntax has become a non-trivial effort recently. Also note that Mizar has a batch-mode proof checker only.

While drawing from the general experience of Mizar, Isar provides a fresh start that avoids these problems. The Isar concepts have been carefully designed with simplicity in mind, while preserving scalability. Isar is based on a lean hierarchic arrangement of basic concepts only. It is quite independent of the underlying logic and its automated tools, by employing a simple meta-logic framework. The Isar/VM interpretation process directly supports interactive development.

Two other systems have transferred Mizar ideas to tactical theorem proving. The \Mizar mode for HOL" [11] is a package that provides means to write tactic scripts in a way resembling Mizar proof text | it has not been developed beyond some initial experiments, though. DECLARE [20, 21] is a much more elaborate system experiment, which draws both from the Mizar and HOL tradition. It has been applied by its author to some substantial formalization of Java operational semantics [21]. DECLARE heavily depends on a its built-in automated procedure for proof checking, causing considerable run-time penalty compared to ordinary tactical proving. Furthermore, proofs cannot be freely arranged according to usual natural deduction practice.

Apart from \declarative" or \intelligible" theorem proving, there are several further approaches to provide human access to formal proof. Obviously, user interfaces for theorem provers (e.g. [3]) would be of great help in performing interactive proofs. Yet there is always a pending danger of overemphasizing advanced interaction mechanisms instead of adding high-level concepts to the underlying system. For example, *proof-by-pointing* offers the user a nice way to select subterms with the mouse. Such operations are rather hard to communicate later, without doing actual replay. In Isar one may select subterms more abstractly via term abbreviations, bound by higher-order matching.

Nevertheless, Isar may greatly benefit from some user interface support, like a *live document* editor that helps the writer to hierarchically step back and forth through the proof text during development. In fact, there is already a prototype available, based on the Edinburgh *ProofGeneral* interface. A more advanced system, should also provide high-level feedback and give suggestions of how to proceed | eventually resulting in *Computer-aided proof writing* (*CAPW*) as proposed in [21]. Furthermore, digestible information about automated proof methods, which are fully opaque in Isar so far, would be particularly useful. To this end, proof transformation and presentation techniques as employed e.g. in ΩMega [2] appear to be appropriate. Ideally, the resulting system might be able to transform internal inferences of proof tools into the Isar proof language format. While this results in bottom-up generation of Isar proofs, another option would be top-down search of Isar documents, similar to the proof planning techniques that ΩMega [2] strongly emphasizes, too. Shifting the focus even more beyond Isar towards actual high-level proof automation, we would arrive at something analogous to the combination of tactical theorem proving and proof planning undertaken in Clam/HOL [4].

We have characterized Isar as being \interpretative" | a higher level language is interpreted in terms of some lower level concepts. In contrast, transformational approaches such as [8] proceed in the opposite direction, abstracting primitive proof objects into a higher-level form, even natural-language.

Most of the Isar concepts presented in this paper have already been implemented within Isabelle. Isar will be part of the forthcoming Isabelle99 release. The system is already sufficiently complete to conduct proofs that are slightly more complex than the examples presented here. For example, a proof of Cantor's theorem that is much more comprehensive than the original tactic script discussed in [16, Part III]. Another example is correctness of a simple translator for arithmetic expressions to stack-machine instructions, formulated in Isabelle/HOL. The Isar proof document nicely spells out the interesting induction and case analysis parts, while leaving the rest to Isabelle's automated proof tools. More realistic applications of Isabelle/Isar are to be expected soon.

References

[1] K. Arkoudas. Deduction vis-a-vis computation: The need for a formal language for proof engineering. The MIT Express project, `http://www.ai.mit.edu/projects/express/`, June 1998.

[2] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMega: Towards a mathematical assistant. In W. McCune, editor, *14th International Conference on Automated Deduction | CADE-14*, volume 1249 of *LNAI*. Springer, 1997.

[3] Y. Bertot and L. Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 11, 1996.

[4] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*. Springer, 1998.

[5] R. Burstall. Teaching people to write proofs: a tool. In *CafeOBJ Symposium, Numazu, Japan*, April 1998.

[6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56{68, 1940.

[7] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, and C. Muñoz. *The Coq Proof Assistant User's Guide, version 6.1.* INRIA-Rocquencourt et CNRS-ENS Lyon, 1996.

[8] Y. Coscoy, G. Kahn, and L. Thery. Extracting text from proofs. In *Typed Lambda Calculus and Applications*, volume 902 of *LNCS*. Springer, 1995.

[9] B. I. Dahn and A. Wolf. A calculus supporting structured proofs. *Journal of Information Processing and Cybernetics (EIK)*, 30(5-6):261{276, 1994. Akademie Verlag Berlin.

[10] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[11] J. Harrison. A Mizar mode for HOL. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'96*, volume 1125 of *LNCS*, pages 203{220. Springer, 1996.

[12] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction | CADE-13*, volume 1104 of *LNCS*, pages 733{747. Springer, 1996.

[13] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 180{192. Springer, 1996.

[14] T. Nipkow and D. v. Oheimb. Java$_{‘ight}$ is type-safe | definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161{170. ACM Press, New York, 1998.

[15] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[16] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[17] L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications*. MIT Press, 1997.

[18] L. C. Paulson. A generic tableau prover and its integration with Isabelle. In *CADE-15 Workshop on Integration of Deductive Systems*, 1998.

[19] P. Rudnicki. An overview of the MIZAR project. In *1992 Workshop on Types for Proofs and Programs*. Chalmers University of Technology, Bastad, 1992.

[20] D. Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, 1997.

[21] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998. Submitted.

[22] A. Trybulec. Some features of the Mizar language. Presented at a workshop in Turin, Italy, 1993.

On the Implementation of an Extensible Declarative Proof Language[⋆]

Vincent Zammit

Sharp Laboratories of Europe, Oxford Science Park, Oxford, UK

Abstract. Following the success of the Mizar [15, 9] system in the mechanisation of mathematics, there is an increasing interest in the theorem proving community in developing similar declarative languages. In this paper we discuss the implementation of a simple declarative proof language (SPL) on top of the HOL system [3] where scripts in this language are used to generate HOL theorems, and HOL definitions, axioms, theorems and proof procedures can be used in SPL scripts. Unlike Mizar, the language is extensible, in the sense that the user can extend the syntax and semantics of the language during the mechanisation of a theory. A case study in the mechanisation of group theory illustrates how this extensibility can be used to reduce the difference between formal and informal proofs, and therefore increase the readability of formal proofs.

1 Introduction

In this paper we illustrate a declarative proof language which we call SPL (standing for Simple Proof Language). We also describe the implementation of a proof checker for SPL on top of the HOL theorem prover [3]. Basically, the declarative language is embedded in ML which is the meta-language of HOL and the proof checker generates HOL theorems from SPL proof scripts. The work described here is illustrated in more detail in the author's thesis [17].

The HOL theorem prover is implemented according to the LCF philosophy, in the sense that:

- HOL theorems are represented by an ML abstract data type whose signature functions correspond to the primitive inference rules of a sound deductive system of the HOL logic. This ensures that theorems derived in the system are valid HOL formulae.
- The user is given the flexibility to implement proof procedures in the meta-language ML in order to facilitate the theorem proving process.
- The HOL system includes a number of ML functions which allow users to find proofs interactively by applying tactics.

Most of the proofs implemented in HOL, and several other proof development systems, are found interactively using the tactic-based goal-oriented environment. The user specifies the required theorem as a goal, and then applies tactics

[⋆] The work presented in this paper was done when the author was a student at the Computing Laboratory of the University of Kent at Canterbury.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 185–202, 1999.

which either solve the goal if it is simple enough, or else break the goal into simpler subgoals. This is repeated until all the subgoals are solved. This mechanism is indeed quite effective for the interactive discovery of proofs because users can use and implement powerful tactics to automate several proof steps, and usually users do not need to remember all the previous steps of interaction during theorem proving. However, a tactic proof is simply a list of steps of interaction which is required to prove a particular theorem on a particular theorem prover, and therefore tactic proofs are not at all informative to a human reader. In general, it is extremely hard to follow, modify or maintain tactic proofs without feedback from the interactive theorem prover.

On the other hand, proofs implemented in the Mizar proof language [15] are easier to follow since they offer more valuable information to a human reader than tactic proofs. The Mizar language is usually described as a declarative proof language, since proof steps state *what* is required to derive a theorem, as opposed to tactic-based procedural proofs which consist of the list of interactions required to derive it, and therefore state explicitly *how* the theorem is proved.

The SPL language described in this paper is declarative and is based on the theorem proving fragment of Mizar. The motivation of this implementation is to experiment with possible ways of increasing the theorem proving power of the language during the mechanisation of a theory. The SPL language is extensible, in the sense that the user can implement new theorem proving constructs and include them in the syntax of the language. Such extensibility is important because theory-specific proof procedures can be implemented which use facts derived during the development of a theory. The Mizar language is not extensible, and extensibility is often claimed to be desirable (see the conclusions of [10]).

The use of such theory-specific proof procedures greatly reduces the length of formal proofs since commonly used sequences of inferences can be automated by such proof procedures. Furthermore, the possibility of extending the proof language during mechanisation can reduce the difference between the size of formal and informal mathematical proofs. One main difference between formal and informal proofs is the level of detail between the two. Formal proofs are usually very detailed while informal proofs often omit proof steps which the intended reader of the proof can easily infer. Another difference which can be noticed is that the proof steps of formal proofs vary considerably in their level of complexity: simple proof procedures which represent trivial inferences are used together with very complex ones which automate several non-trivial inferences. As a result, the use of theory-specific proof procedures can be used to automate the proof steps that are usually considered to be trivial by the authors of informal proofs.

Our work is in some respect similar to that done by Harrison [5] who implemented a Mizar mode in HOL. This mode is, however, very much based on the tactic-based environment in HOL since Mizar proof constructs are translated into HOL tactics. The SPL language is richer than the Mizar mode in HOL since, for instance, SPL scripts can be structured into sections to allow a more modular presentation. The processing of SPL scripts is not based on HOL tactics.

Recently, Syme [12] has developed a declarative proof language, DECLARE, for software verification and used it to verify the type correctness of Java [13, 14]. This language is, however, not extensible, although this is suggested in the future work section of [12]. The Mizar mode described in [5] allows the use of arbitrary HOL tactics for justifying proof steps, and is therefore extensible.

In the following sections, we first illustrate the constructs of SPL and then discuss the implementation of the proof checker on top of the HOL theorem prover in Sec. 3, which is followed by a section on the proof procedures used to support the proof checking process. In Sec. 5 we illustrate a user-extensible database of knowledge which can be used to store and derive SPL facts that are considered to be trivial. A case study involving the mechanisation of group theory in SPL is described in Sec. 6, and Sec. 7 gives some concluding remarks and directions for future work.

2 SPL: A Simple Proof Language

The SPL language is based on the theorem proving fragment of Mizar. The language is embedded in ML and the SPL proof checker generates HOL theorems from SPL scripts, and therefore the proof checker is *fully-expansive*: all theorems are derived by the primitive rules of the HOL core inference engine in order to minimise human errors in the proofs. One can also use HOL definitions, axioms, theorems and also proof procedures in SPL proof scripts. No SPL constructs are defined for introducing definitions in a HOL theory, instead one can use the HOL definition packages.

In this section we first use a simple example to describe briefly the syntax of SPL, and then describe the various constructs of the language. A more elaborate treatment is given in Chap. 4 of [17].

Figure 1 illustrates a fragment of an SPL proof script which derives the following theorems:

$$\begin{aligned}\texttt{R\_refl} = \vdash\ & \forall R.\ \texttt{Symmetric}\ R \Rightarrow \texttt{Transitive}\ R \Rightarrow \\ & (\forall x.\ \exists y.\ R\ x\ y) \Rightarrow \texttt{Reflexive}\ R\end{aligned}$$

$$\begin{aligned}\texttt{R\_equiv} = \vdash\ & \forall R.\ \texttt{Symmetric}\ R \Rightarrow \texttt{Transitive}\ R \Rightarrow \\ & (\forall x.\ \exists y.\ R\ x\ y) \Rightarrow \texttt{Equivalence}\ R\end{aligned}$$

where the predicates `Reflexive`, `Symmetric`, `Transitive` and `Equivalence` are defined as follows:

$$\vdash_{def} \forall R.\ \texttt{Reflexive}\ R \equiv (\forall x.\ R\ x\ x)$$

$$\vdash_{def} \forall R.\ \texttt{Symmetric}\ R \equiv (\forall x\ y.\ R\ x\ y = R\ y\ x)$$

$$\vdash_{def} \forall R.\ \texttt{Transitive}\ R \equiv (\forall x\ y.\ R\ x\ y \Rightarrow \forall z.\ R\ y\ z \Rightarrow R\ x\ z)$$

$$\begin{aligned}\vdash_{def} \forall R.\ & \texttt{Equivalence}\ R \equiv \\ & (\texttt{Reflexive}\ R \wedge \texttt{Symmetric}\ R \wedge \texttt{Transitive}\ R)\end{aligned}$$

```
section on_symm_and_trans

  given type ":’a";
  let "R:’a ! ’a ! bool";
  assume R_symm:  "Symmetric R"
         R_trans: "Transitive R"
         R_ex:    "8x. 9y. R x y";

  theorem R_refl: "Reflexive R"
  proof
    simplify with Reflexive, Symmetric and Transitive;
    given "x:’a";
    there is some "y:’a" such that
         Rxy: "R x y" by R_ex;
       so Ryx: "R y x" by R_symm, Rxy;
    hence "R x x" by R_trans, Rxy, Ryx;
  qed;

  theorem R_equiv: "Equivalence R"
        <Equivalence> by R_refl, R_symm and R_trans;

end;
```

Fig. 1. An Example SPL Proof Script.

The proofs of the two theorems are implemented in an SPL section which is called on_symm_and_trans. This section starts at the section keyword and is closed by the end; on the last line. Sections are opened in order to declare *reasoning items*, which include the introduction of assumptions, the declaration and proof of theorems, etc.

The first two reasoning items in this section introduce the type variable :’a and the variable R so that they can be used in later reasoning items. A number of assumptions are then introduced (locally to this section) which are used by the proofs of the two theorems. Theorems are declared by the theorem keyword which is followed by an optional label, the statement of the theorem and its *justification*. The justification of the first theorem is a proof which consists of a number of steps, while the simple justification of the second theorem consists of a single line.

The simplify statement in the proof of the theorem R_refl is a *simplifier declaration*. A simplifier is a proof procedure which simplifies the statement of assumptions, theorems, and proof step results, and can be declared using `simplify` constructs so that it can be used automatically during proof checking. Definitions can be declared as simplifiers so that they are unfolded automatically. The expression `<Equivalence>` in the justification of the second theorem is a simplifier declaration which is local to that simple justification.

The theorems derived in the script given in Fig. 1 can still be used outside section `on_symm_and_trans`, however their statements are *expanded*, or generalised, according to the variables and assumptions local to this section, that is to the statements given earlier this section.

2.1 On SPL Sections

The sectioning mechanism of SPL is similar to that of the Coq system [1]. In general, an SPL proof script consists of a list of sections, and sections can be nested to improve the overall structure of scripts. All the information declared in a section is local to it, and (with the exception of theorems) local information is lost when a section is closed. Theorems proved in a section are still visible outside it, however they are generalised according to the local variables and assumptions used in deriving them. The advantages of declaring information locally can also be seen in the simple example given earlier in Fig 1. In particular, the statements of the theorems declared in the proof script are shorter than their fully expanded form given in the beginning of this section, and therefore:

- Repetitive information in the statements of theorems is avoided, for instance the antecedents of the two theorems in our example are declared once as the assumptions local to both theorems.
- The unexpanded form of the statement of theorems in the section in which they are derived is due to the fact that they are specialised by the information declared locally, which includes the generalising variables and assumptions. As a result, justifications using unexpanded theorems do not have to include the assumptions which are used in deriving them. For example, when the theorem `R_refl` is used in justifying the theorem `R_equiv`, there was no need to include the three assumptions used in deriving `R_equiv`. As a result, justifications which use unexpanded results are shorter, and also easier to proof check, than those which use the results in their fully generalised form.
- Since proof statements and proofs are shorter, scripts are easier to read.

2.2 Reasoning Items

SPL reasoning items include generalisation which introduce new type variables and (term) variables, introductions of assumptions, abbreviations (local definitions) which allow expressions to be represented by a single word, and the declaration of simplifiers. Reasoning items also include the declaration and justification of *facts* which include theorems and intermediate results (such as `Rxy` and `Ryx` in Fig. 1).

2.3 Justifications

The statements of intermediate proof step results and theorems are followed by their justifications which include *straightforward justifications* usually consisting of the `by` keyword followed by a number of premises. An optional *prover* can be specified before the list of premises; a prover represents a HOL decision procedure which derives the conclusion of the justification from its premises. A default prover is assumed if none is specified. As shown in the proof of the second theorem in Fig. 1, one can declare an optional local list of simplifiers before the `by` keyword. Justifications can also be of the form of proofs which consist of a list of intermediate results and other reasoning items between a `proof` and a `qed` or `end` keyword. A commonly used type of justification is the iterative equality such as the following which justifies the fact `a + (b + c) = (c + a) + b` labelled with `abc`:

```
abc: "a + (b + c)  =  a + (c + b)" by commutativity
                ." =  (a + c) + b" by associativity
                ." =  (c + a) + b" by commutativity;
```

2.4 SPL Sentences

SPL sentences are the expressions in the syntax of SPL which denote facts (which include theorems, assumptions and intermediate results). Usually sentences consist simply of the label of the fact. However, one can specify a list of local simplifiers which are applied to a single fact during proof checking. One can also specify a forward inference rule to derive facts from other facts, variables and some other expressions. The use of local simplifiers and forward inference rules is however not encouraged because of the procedural nature of the resulting proofs.

3 The Implementation of an SPL Proof Checker

The proof checker of the SPL language implemented in HOL processes proof scripts in two steps:

{ Parsing the input text into an internal (ML) representation of the language constructs;

{ Processing the constructs to modify the environment of the proof checker.

The SPL state is represented by an ML object of type `reason_state` and consists of the input string and the environment of type `reason_environment`. The implementation of the proof checker consists of a number of ML functions which parse and process SPL constructs. Such functions take and return objects of type `reason_state`. A number of other functions which act on objects of type `reason_state` are also implemented. These include functions which extract proved theorems from the SPL environment so that they can be used in HOL, add HOL axioms, definitions and theorems to the environment, and add new input text in order to be parsed and processed.

The processing of SPL scripts can therefore be invoked during a HOL theorem proving session by calling the appropriate ML functions. As a result, the user can implement an SPL script, process it within a HOL session and use the derived results in HOL inference rules and tactics or in the implementation of proof procedures in ML. Moreover, the SPL language is extensible: the user can implement HOL proof procedures and include them in the language syntax. Therefore, one can develop a theory by repeating the following steps:

(i) deriving a number of theorems using SPL proofs,
(ii) using the derived theorems in the implementation of HOL proof procedures,
(iii) extending the SPL language to make use of the new proof procedures.

This approach combines the readability of SPL proofs with the extensibility of the HOL system. The mechanisation of group theory described briefly in Sec. 6 is developed using this approach. In this case, new proof procedures were implemented as the theory was mechanised in order to automate the proof steps which would be considered trivial by the reader.

ML references are used to store the functions which parse and process the SPL language constructs (including the parser and processors of reasoning items) so that they can be updated by the user during the development of a theory. This simply mechanism allows the SPL parser (which consists of the collection of the functions which parse the language constructs) to be user-extensible. Provers (which correspond to HOL decision procedures), simplifiers, and forward inference rules are also stored in ML references. This implementation design was originally used to allow the author to alter the syntax and semantics of the language easily during the development of a theory when the implementation of the SPL language was still in its experimental stages. However, we now believe that the flexibility and extensibility offered by this design can indeed be a desirable feature of proof languages. This allows the proof implementor, for instance, to include new reasoning items (rather than just proof procedures) which make use of derived theorems during the implementation of a theory. One can also change substantial parts of the syntax of the language to one which he or she believes to be more appropriate to the particular theory being mechanised. Ideally, any alterations made to the syntax of the language should be local to particular sections. In order to achieve this, one needs a number of design changes to the current implementation of the language since the use of ML references allows the user to update the syntax globally rather than locally. A number of ML functions are implemented in order to facilitate the update of the parsing and processing functions stored in ML references.

The object embedding system of Slind [11] is used to embed the SPL language in SML. Basically, using this system the text of SPL scripts and script fragments is enclosed in backquotes (‘) so that they can be easily written and read. The texts are however internally represented as ML objects from which ML strings representing the lines of the proof texts can be extracted. Once extracted the strings are then parsed using the SPL language parser. The SPL language uses the HOL syntax for terms and types. SPL expressions representing terms and types are given to the internal HOL parser after a simple preprocessing stage which, for instance, gives the type `:bool` to expressions representing formulae, and inserts types for any free variables which have been introduced by generalisations.

Because of the hierarchical structure of SPL scripts, the SPL environment (which represents the information that has been declared and derived by SPL constructs) is structured as a stack of *layers* containing the information declared locally. An empty layer is created and pushed on top of the stack at the beginning of a section or proof. Processing reasoning items affects only the information in the top layer. At the end of a section or proof, the top layer is popped from the stack and all the information stored in this layer, with the exception of theorems, is destroyed. Theorems are expanded and inserted into the new top layer. The expansion of theorems involves the substitution of local abbreviations with the terms they represent, the discharging of locally introduced assumptions, and the generalisation of locally introduced variables. We say that a layer has been opened when it is pushed on top of the environment stack. We also say that a layer has been closed when it is popped from the stack.

Each layer contains a list of locally derived or assumed facts labelled by their identifier, a list of variables and type variables introduced by reasoning items, a list of declared simplifiers, and some other information (e.g., the name of the section, the current conclusion in case of a proof layer, etc.).

4 Support for Automated Proof Discovery

The proof checking process of SPL scripts, which involves the generation of HOL theorems from SPL constructs, is supported by a number of proof procedures, which include:

Inference Rules which allow the user to derive facts in a procedural manner using any forward inference rule. The use of these rules is not encouraged because it may reduce the readability of proof scripts.

Simplifiers which can be used to normalise terms, and to perform calculations which would be considered trivial in an informal proof. Any HOL conversions can be included by the user as SPL simplifiers.

Proof Search Procedures or simply provers, which are used to derive the conclusions of straightforward justifications from a given list of premises.

The user can implement any of these kinds of proof procedures in ML during the development of a theory, associate SPL identifiers with them, and include them in the syntax of the language.

The SPL implementation includes a knowledge database which can be used to store facts which are considered to be trivial. This database can be queried by any of the above kinds of proof procedures in order to obtain trivial facts automatically. The use of this database is described in the next section.

Only one forward inference rule is used in the mechanisation of group theory described in Sec. 6. This rule corresponds to the introduction of the Hilbert choice operator and takes a variable v and a sentence denoting some fact $P[v]$ and derives $P[\text{"}v{:}P[v]]$.

Several simplifiers have been implemented during this case study in order to extend the SPL language with group theory specific proof procedures. The main role of the simplifiers used in the case study is to normalise certain expressions (such as group element expressions involving the identity element, the group product, the inverse function, and arbitrary group elements) once normal forms have been discovered. A number of mathematical theories are *canonisable*, that is, their terms can be uniquely represented by a canonical, or normal form. Theories whose terms can be normalised effectively have a decidable word problem since two terms are equal if and only if their respective normal forms are syntactically identical. Therefore theory-specific normalisers can be used to derive the equality of certain terms automatically. It should be noted, that once normal forms are discovered and described in informal mathematical texts, the simplification of a term into its normal form is usually considered to be obvious and omitted from proofs. The use of simplifiers to automate the normalisation process can therefore reduce the difference between formal and informal proofs.

The default prover used in the case study is a semi-decision procedure for first-order logic with equality implemented as a HOL derived rule. The proof search process involves the discovery of a closed tableau and uses rigid basic superposition with equational reflexivity to reason with equality [2]. This semi-decision procedure is complete for first-order logic with equality, however a number of resource bounds are imposed on the proof search process since the straightforward justifications of SPL scripts usually represent rather trivial inferences[1]. The implementation of this decision procedure is discussed in Chap. 5 of [17].

Since SPL formulae are higher-order, they need to be transformed into first-order ones before they can be used by the above mentioned tableau prover. This is done by:

1. Normalising them into β-long η normal form,
2. Eliminating quantification over functions and predicates by the introduction of a new constant $@ : (y \to \delta) \to y \to \delta$ ($@$ for "apply") and then transforming terms of the form $(f\ x)$ into $(@\ f\ x)$ so that higher-order formulae like $\forall P.\ P\ x \Rightarrow P\ y$, are transformed into first-order ones (like $\forall P.\ @\ P\ x \Rightarrow @\ P\ y$).
3. Eliminating lambda abstractions, which for instance transforms the formula:

   $$(a = b) \Rightarrow P\ (\lambda x.\ f\ x\ a) \Rightarrow P\ (\lambda x.\ f\ x\ b)$$

   into the valid first-order formula

   $$(a = b) \Rightarrow P\ (g\ f\ a) \Rightarrow P\ (g\ f\ b)$$

   with the introduction of the constant $g = (\lambda y, z, x.\ y\ x\ z)$.

[1] The tableau prover mentioned in this section is modified to proof check a special kind of justifications which we call *structured straightforward justifications*. These justifications are used in the implementation of the case study mentioned in Sec. 6. We do not discuss this type of justifications in this paper. The reader is referred to Chapters 6 and 8 of [17].

5 A Database of Trivial Knowledge

As mentioned in the introduction of this paper, one major difference between formal and informal proofs is the level of detail between the two. Informal proofs contain gaps in their reasoning which the reader is required to fill in order to understand the proof. The author of an informal proof usually has a specific type of reader in mind: one who has a certain amount of knowledge in a number of mathematical fields, and one who has read and understood the preceding sections of the literature containing the proof. The author can therefore rely on his, usually justified, assumptions about what the intended reader is able to understand when deciding what to include in an informal proof and what can be easily inferred by the reader, and can (or must) therefore be unjustified. For example, if one assumes that some set A is a subset of B, and that some element a is a member of A, then the inference which derives the membership of a in B can usually be omitted if the reader is assumed to be familiar with the notions of set membership and containment. On the other hand, it is very often the case that when a substantial fragment of a theory has been developed using a theorem proving environment, the formal proofs may still contain inferences which use trivial results that have been derived much earlier in the mechanisation.

Since the need to include explicitly such trivial inferences in most formal proof systems results in the observed difference between the size and readability of formal and informal proofs, we have experimented with the implementation of a simple user-extensible knowledge database which proof procedures can query in order to derive trivial facts automatically.

The knowledge in the database is organised into *categories* each containing a list of facts. New categories can be added during the development of a theory. For example, in order to derive the trivial inference illustrated in the example given earlier this section, one can include a membership category with identifier in_set in order to include facts of the form *x is a member of X*, and a containment category subset which includes facts of the form *X is a subset of Y*. SPL facts can then be stored in the database during proof implementation using the construct:

```
consider in_set a is a member of A
         subset A is a subset of B ;
```

In order that these facts can be used by proof procedures, the user is also required to implement ML functions which query the database. Such functions take the knowledge database as an argument together with a number of other arguments depending on the category they query. For example, a function to query the in_set category may take a pair of terms representing an element and a set. Query functions return a theorem when they succeed. ML references can be used to store the searching routine of the query function so that it can be updated during the development of a theory, as shown in the SML fragment in Fig. 2. The user can then implement proof procedures (such as simplifiers) which call this query function.

```
fun in_set_search kdbs (e, s) =
   look for the fact “e is in s" in kdbs
   and return it if found;
   otherwise raise an exception

local
  (* store the search function in a reference *)
  val in_set_ref = ref in_set_search
in

  (* the query calls the stored search function: *)
  fun in_set kdbs query = (!in_set_ref) kdbs query

  (* updating the query function *)
  fun update_in_set new_qf =
    let val old_in_set = !in_set_ref
        fun new_in_set kdbs query =
              old_in_set kdbs query (* try the old query: *)
              handle _ =>           (* if it fails *)
                new_qf kdbs query   (*    try the new one: *)
     in in_set_ref := new_in_set    (* update the store function: *)
    end

end;
```

Fig. 2. The Implementation of a Query Function.

Query functions can also be implemented to handle existential queries. For example an existential query function for the subset category can take a set X as an argument and looks for a fact of the form X *is a subset of* Y for some set Y. A different existential query function on the same category would look for some fact Y *is a subset of* X. Since many such facts may be derived by the knowledge database, existential query functions are implemented to return a lazy sequence of facts satisfying the query.

Query functions can be updated when new results are derived which can be used in the automatic deduction of trivial facts. For example, given the derived fact

$$\forall x; X; Y: \ (x \text{ is in } X) \Rightarrow (X \text{ is a subset of } Y)$$
$$\Rightarrow (x \text{ is in } Y)$$

one can then update the query function of `in_set` so that given some query *a is in B* it

1. calls the appropriate existential `subset` query function to check whether there is some set *A* such that *A is a subset of B* can be derived from the database, and
2. queries `in_set` (recursively) to check whether *a is in A* for some *A* satisfying the previous query.

Given the required facts, the new `in_set` query function can then derive and return the fact *a is in B* using the above result. As the search function is stored in an ML reference, updating a query function affects the behaviour of all the proof procedures which use it.

Since some search is needed in the handling of most queries, and since the same query may be made several times during theorem proving, the output of successful non-existential queries is cached to avoid repeated search. In the current implementation caches are stored globally and are reset when a layer containing knowledge which can affect the query concerned is closed. A better approach would be to store caches locally in each layer.

Case studies involving the implementation of formal proofs in SPL showed that the length of the proofs can be substantially reduced through the use of a knowledge database. This reduction of proof length is due to the implementation of theory-specific query functions which make use of derived theorems, as well as the implementation of proof procedures which are able to query the database. We notice that the implementation of such functions with the intention of minimising the difference between formal and informal proofs involves the understanding of what authors of informal proofs consider to be trivial by the intended reader. Therefore, the implementation of functions capable of deriving facts which are considered to be trivial by a knowledgeable reader is a formal means of illustrating what can be considered obvious in some particular proof and how such obvious facts can be derived. We argue that this is a formal means of representing a particular kind of knowledge and understanding in a mathematical field other than giving a list of detailed formal proofs. We believe that the presentation of such information should be included in a formal development of a mathematical field.

In the case study discussed in Sec. 6, the only proof procedures which use the knowledge database are the simplifying procedures. The main reason for this is the fact that the proof search procedures were implemented before the experimental database was designed. However, in principle the proof procedures can be redesigned and implemented to be able to query the database. We will consider this area for future work and believe that the length of formal proofs can be greatly reduced with such a feature.

6 A Mechanisation of Group Theory: A Case Study

In this section we describe a case study involving the mechanisation in SPL of a number of results in group theory. The mechanisation is based on the textbook by Herstein [6] and includes results on normal groups, quotient groups and the isomorphism theorems. Due to space limitations, we only give a brief outline of the mechanisation here. A more detailed description is given in Chap. 9 of [17]. Since one of the motivations of this case study is to experiment with a user-extensible declarative proof language in order to reduce the difference between formal and informal proofs, the mechanisation includes the implementation of several group theory specific proof procedures in ML. These proof procedures include a number of simplifiers and several query functions on the SPL knowledge database (Sec. 5). These proof procedures are used to automate a number of the inferences that are omitted from the proofs in [6].

We remark that the mechanisation of group theory in a theorem proving environment is not a novel idea. In particular, all the results formalised in this case study (and several others) have been formalised in the Mizar system, and Gunter formalised a number of results on group theory in HOL [4]. The contribution of this case study lies in the use of an extensible declarative proof language in which proof procedures are implemented to improve the readability of the proof scripts.

6.1 Group Theory in the HOL Logic

Groups are defined in the simply typed polymorphic logic of HOL in a similar fashion to the definitions in [4, 7] as a pair $(G;p)$ satisfying the group axioms where G is a polymorphic predicate representing the set of group elements and p is a binary operator on the elements in G. The predicate Group $(G;p)$ defined below holds if $(G;p)$ is a group:

```
'def Group (G:   ! bool; p:   !   !   )
        (GClosed (G;p)) ^
        (GAssoc (G;p)) ^
        9e: . (G e) ^ (GId (G;p) e) ^
              (8x. G x ) GhasInv (G;p) e x)
```

where GClosed $(G;p)$ and GAssoc $(G;p)$ hold if G is closed under p, and p is associative on the elements of G respectively. The formula GId $(G;p)$ e holds if e is the left and right identity of p in G, GhasInv $(G;p)$ e x holds if there is some y such that GInv $(G;p)$ e x y holds, which holds if y is some left and right inverse of x in $(G;p)$ assuming that e is an identity element in G. This definition corresponds to the definition of groups given in [6].

Given a group $(G;p)$, an identity element can be selected by the function IdG, and given an element in G, its inverse can be selected by the function InvG; these functions are defined as follows:

```
'def IdG (G;p)    "e.G e ^ GId (G;p) e

'def InvG (G;p) x    "x1.G x1 ^ GInv (G;p) (IdG (G;p)) x x1
```

6.2 Preliminary Results

In the textbook by Herstein [6], once the above definition of a group is introduced, a number of simple but quite useful results have been defined. These results include the uniqueness of the identity and inverse elements, and some simplifications on group element expressions such as $(a^{-1})^{-1} = a$ and $(ab)^{-1} = b^{-1}a^{-1}$ for all group elements a and b, where x^{-1} represents the inverse element of x and the juxtaposition of two group elements represents their product.

Once derived, such results are then assumed to be obvious in the text: they are inferred without any justification. In order to achieve the same effect in the formal SPL script, one has to implement proof procedures which automate the inference which are assumed obvious in the informal text. In this case study a simplifier which normalises expressions representing group elements is implemented after groups are defined. This normaliser is based on the following rewrite rules which correspond to a strongly normalising term rewriting system for group element expressions generated by the inverse function, the group product and the identity element e (see [8] for e.g.,).

$$\begin{array}{rcl@{\qquad}rcl}
ex & \to & x & xe & \to & x \\
(x^{-1})x & \to & e & x(x^{-1}) & \to & e \\
(xy)z & \to & x(yz) & (x^{-1})^{-1} & \to & x \\
e^{-1} & \to & e & (xy)^{-1} & \to & y^{-1}x^{-1} \\
x(x^{-1}y) & \to & y & x^{-1}(xy) & \to & y
\end{array}$$

These rules are derived using declarative proofs in SPL as HOL theorems and used in the implementation of a simplifier (with identifier groups) for group elements. It should be noted that because of the simple type theory the HOL logic is based on, the theorems corresponding to the above rules contain a number of antecedents corresponding to the facts that the elements in the rules are members of the same group. For example, the rule corresponding to the associativity of the group product is given by the theorem:

$$\begin{array}{l}
\vdash \forall G\ p.\ \mathtt{Group}\ (G,p) \Rightarrow (\forall x\ y\ z.\ G\ x \Rightarrow G\ y \Rightarrow G\ z \Rightarrow \\
\qquad (p\ (p\ x\ y)\ z = p\ x\ (p\ y\ z)))
\end{array}$$

In order to derive the antecedents of such rules automatically, SPL knowledge databases for the Group predicate and set membership are introduced, and the groups simplifier is implemented so that it queries the database to check whether all the antecedents of a rule hold before it is applied. Since a query to the knowledge database can correspond to several inferences (such as the use of the transitivity of the subset relation in deriving whether an element is a member of some set), several proof steps can be omitted if the groups simplifier is used. Given this simplifier, the left cancellation law is derived in SPL as follows:

```
theorem Cancel_left : "(p z x = p z y) ⇒ (x = y)"
proof
  assume zx_eq_zy: "p z x = p z y";
```

```
   "x = p (inv z) (p z x)" <groups> by fol
    ."= p (inv z) (p z y)" by zx_eq_zy
    ."= y" <groups> by fol;
  qed;
```

once the facts that x, y and z are in G, and that (G, p) is a group are assumed and stored in the knowledge database. The justification `<groups> by fol` states that `groups` is used as a local simplifier and that the first-order logic prover (`fol`) is used as the proof search procedure. The term `inv` is a local abbreviation to the expression which corresponds to the inverse function in the group (G, p).

The database categories and their query functions were updated whenever definitions were introduced and new results were derived in SPL. For example, when subgroups were defined, the query function for the group category was updated so that the fact that H is a group is automatically derived if H is stated to be a subgroup of some group G; and the set membership category was updated so that the fact that the identity element of G is in its subgroup H is automatically derived when the equality of the identity elements of H and G was proved.

6.3 Cosets, Normal Subgroups, and Further Results

The procedure of deriving theorems in SPL and then using them to implement and update simplifiers and database query functions to extend the SPL language and to use these proof procedures in later SPL proofs to derive more results, is repeated throughout the mechanisation of group theory. For example, when left and right cosets and products of subsets of groups are defined:

$$\vdash_{def} \texttt{RightCoset}\ (H, p)\ a \equiv (\lambda b.\ \exists h.\ H\ h \wedge (b = p\ h\ a))$$

$$\vdash_{def} \texttt{LeftCoset}\ a\ (H, p) \equiv (\lambda b.\ \exists h.\ H\ h \wedge (b = p\ a\ h))$$

$$\vdash_{def} \texttt{SProd}\ p\ X\ Y \equiv (\lambda x.\ \exists h.\ X\ h \wedge \exists k.\ Y\ k \wedge (x = p\ h\ k))$$

a simplifier `cos` is implemented which normalises terms involving these expressions. Cosets are denoted in the informal literature by juxtapositioning the group subset with the group element, and the product of two subsets is denoted by juxtapositioning the two subsets.

Similarly to the `groups` simplifier, the `cos` simplifier queries the database to derive certain facts automatically. This simplifier is used, for example, in a very famous result in group theory which states that if N is a *normal* subgroup of some group G (i.e., if $N = gNg^{-1}$ for all $g \in G$), then $(Na)(Nb) = Nab$ for all a and b in G. This result is derived by the equations:

$$(Na)(Nb) = N(aN)b = N(Na)b = NNab = Nab$$

and uses the facts that if N is a subgroup then $NN = N$ and if N is normal then $Na = aN$. This result is derived in SPL by the same sequence of steps with the help of the `cos` simplifier enhanced with appropriate database query functions:

```
"SProd p (RightCoset (N,p) a) (RightCoset (N,p) b)
    = SProd p N (RightCoset (LeftCoset a (N,p),p) b)"<cos> by fol
  ."= SProd p N (RightCoset (RightCoset (N,p) a,p) b)"
          by Normal_gN_Ng, Ga
  ."= RightCoset (SProd p N N,p) (p a b)"<cos> by fol
  ."= RightCoset (N,p) (p a b)" by SProd_Idem, GroupG, NsgG;
```

where `Normal_gN_Ng` is the theorem stating that $gN = Ng$ for g 2 G, `Ga` is the fact a 2 G, `GroupG` is the fact that $(G; p)$ is a group, and `NsgG` is the fact that N is a subgroup of G. The theorem `SProd_Idem` states that the product HH of a subgroup H is equal to H.

The mechanisation of group theory in SPL derives all the results in Herstein [6] up to and including the second isomorphism theorem with the exception of those involving finite groups. Simplifiers (other than `groups` and `cos`) which query the database of trivial knowledge are implemented in order to minimise the difference between the SPL proofs and those found in the literature. A more detailed account of this mechanisation is given in [17].

7 Conclusions and Future Work

In this paper we have illustrated an extensible declarative proof language, SPL, which is very much based on the theorem proving fragment of the Mizar proof language. A fully-expansive proof checker for SPL is implemented on top of the HOL theorem proving system. The extensibility of the language is achieved by storing the ML functions which parse and process SPL scripts in ML references so that they can be updated during the mechanisation of a theory. The proof checker is supported by a number of proof procedures which include simplifiers, and a tableau-based prover for first-order logic with equality. These proof procedures are also stored in ML references so that they can be updated during theory mechanisation. A user-extensible database of SPL facts is used to derive results which are considered trivial and should therefore be implicit during proof implementation. The SPL proof procedures can query this database to derive such results automatically. Finally, a sectioning mechanism, similar to that of the Coq system, is used in order to give a hierarchical structure to SPL scripts.

A case study involving the mechanisation of group theory in SPL is also described briefly in this paper. The extensibility of the SPL language is used so that simplifiers and database query functions are implemented in order to derive certain facts automatically whose proof is usually omitted in the informal mathematical literature. As a result, the proofs implemented in this mechanisation are quite similar (in the number and complexity of proof steps) to those found in the literature. This case study shows that the extensibility of a declarative proof language is indeed a powerful feature which results in more readable proof scripts.

We remark that it was possible to implement the SPL proof checker on top of the HOL system because of the way the HOL system is designed. In particular,

1. a Turing-complete metalanguage is available to allow the user to extend the system with new proof procedures and proof environments, and
2. the fact that all HOL theorems are constructed using the core inference engine ensures that such extensions are safe.

It is possible to implement proof checkers of declarative languages such as SPL on top of other theorem proof environments if they provide these two features.

We note that although a number of proofs mechanised during the case study are observed to be similar to their informal counterparts when the number of *steps* in the proofs are compared, the length of the *symbols* in the formal proofs is still much higher than that of the informal proofs. The authors of informal mathematics very often change the syntax of their language by introducing appropriate notations. It is therefore desirable that one is able to safely modify the term parser of the proof checker during the mechanisation of a theory, and such a possibility needs to be explored in future.

Also, in our case study, the only proof procedures which query the knowledge database are the simplifiers. Possible future work involves the implementation of proof search procedures (i.e., decision procedures) which can query the database. Unfortunately, query database functions are implemented manually in ML, and the possibility of designing and using some higher-leverl language should be considered. The SPL language can also be extended by the introduction of theory specific reasoning items; this feature is not exploited in our case study, and therefore other case studies on mechanisations involving the use of such extensibility are required in order to evaluate their effect in practice.

8 Acknowledgements

I would like to thank my supervisor, Simon Thompson, for his supervision and his continuous support on the work presented here, and throughout the three year period of my PhD. I thanks the anonymous referess for their very helpful and interesting remarks on this paper. I would also like to thank Toshio Nomura of Sharp Labs. of Europe for giving me the possibility to write this paper.

References

[1] Bruno Barras et al. *The Coq Proof Assistant Reference Manual.* Projet Coq | INRIA-Rocquencourt, CNRS-ENS Lyons, November 1996. (Version 6.1).

[2] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid E-unification. *Journal of Automated Reasoning*, 20(1):47{80, 1998.

[3] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[4] Elsa Gunter. The implementation and use of abstract theories in HOL. In *Proceedings of the Third HOL Users Meeting*, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark, October 1990. Technical Report DAIMI PB { 340 (December 1990).

[5] J. Harrison. A Mizar mode for HOL. In von Wright et al. [16], pages 203{220.

[6] I.N. Herstein. *Topics in Algebra*. John Wiley & Sons, New York, 2nd edition, 1975.

[7] L. Laibinis. Using lattice theory in higher order logic. In von Wright et al. [16], pages 315{330.

[8] David A. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, J. A. Robinson, and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 5, pages 273{364. Oxford University Press, Oxford, 1993.

[9] Piotr Rudnicki. An overview of the MIZAR project. Available by ftp from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z.`, June 1992.

[10] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. Available on the web at `http://www.cs.ualberta.ca/~piotr/Mizar/Wfnd/`, January 1997.

[11] Konrad Slind. Object language embedding in Standard ML of New Jersey. In *Proceedings of the Second ML Workshop held at Carnegie Mellon University, Pittsburgh, Pennsylvania, Septermber 26-27, 1991, CMU SCS Technical Report*, November 1991.

[12] Don Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.

[13] Don Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.

[14] Donald Syme. *Declarative Theorem Proving for Operating Semantics*. PhD thesis, University of Cambridge, 1998. Submitted for Examination.

[15] A. Trybulec. The Mizar-QC/6000 logic information language. *Bulletin of the Association for Literary and Linguistic Computing*, 6:136{140, 1978.

[16] J. von Wright, J. Grundy, and J. Harrison, editors. *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, August 1996. Springer.

[17] Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent at Canterbury, October 1998. Submitted for Publication. (Currently available on the world wide web at the url: `http://www.cs.ukc.ac.uk/people/rpg/vz1/thesis.ps.gz`).

Three Tactic Theorem Proving

Don Syme

Microsoft Research Limited, St. George House, 1 Guildhall Street, Cambridge, CB2 3NH, UK

Abstract. We describe the key features of the proof description language of Declare, an experimental theorem prover for higher order logic. We take a somewhat radical approach to proof description: proofs are not described with tactics but by using just three expressive outlining constructs. The language is \declarative" because each step specifies its logical consequences, i.e. the constants and formulae that are introduced, independently of the justification of that step. Logical constants and facts are lexically scoped in a style reminiscent of structured programming. The style is also heavily \inferential", because Declare relies on an automated prover to eliminate much of the detail normally made explicit in tactic proofs. Declare has been partly inspired by Mizar, but provides better automation. The proof language has been designed to take advantage of this, allowing proof steps to be both large and controlled. We assess the costs and benefits of this approach, and describe its impact on three areas of theorem prover design: specification, automated reasoning and interaction.

1 Declarative Theorem Proving

Interactive theorem provers combine aspects of formal specification, manual proof description and automated reasoning, and they allow us to develop machine checked formalizations for problems that do not completely succumb to fully automated techniques. In this paper we take the position that the role of proof description in such a system is relatively simple: it must allow the user to describe how complex problems decompose to simpler ones, which can, we hope, be solved automatically.

This article examines a particular kind of *declarative proof*, which is one technique for describing problem decompositions. The proof description language we present is that of Declare, an experimental theorem prover for higher order logic. The language provides the functionality described above via three simple constructs which embody first-order decomposition, second-order proof techniques and automated reasoning. The actual implementation of Declare provides additional facilities such as a specification language, an automated reasoning engine, a module system, an interactive development environment (IDE), and other proof language constructs that translate to those described here. We describe these where relevant, but focus on the essence of the outlining constructs.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 203{220, 1999.

In this section we describe our view of what constitutes a declarative proof language and look at the pros and cons of a declarative approach. We also make a distinction between \declarative" and \inferential" aspects of proof description, both of which are present in the language we describe. In Section 2 we describe the three constructs used in Declare, and present a longer example of proof decomposition, and Section 3 discusses the language used to specify hints. Section 4 compares our proof style with tactic proof, and summarizes related issues such as automated reasoning and the IDE.

Space does not permit extensive case studies to be presented here. However, Declare has been applied to a formalization of the semantics of a subset of the Java language and a proof of type soundness for this subset [Sym99]. The purpose of Declare is to explore mechanisms of specification, proof and interaction that may eventually be incorporated into other theorem proving systems, and thus complement them.

1.1 Background

This work was inspired by similar languages developed by the Mizar group [Rud92] and Harrison [Har96]. Mizar is a system for formalizing general mathematics, designed and used by mathematicians, and a phenomenal amount of the mathematical corpus has been formalized in this system. The foundation is set theory, which pervades the system, and proofs are expressed using detailed outlines, leaving the machine to fill in the gaps. Once the concrete syntax is stripped away, steps in Mizar proofs are mostly applications of simple deduction rules, e.g. generalization, instantiation, and propositional introduction and elimination.[1] Essentially our work has been to transfer a key Mizar idea (proof outlining) to the setting of higher order logic theorem proving, use extensive automation to increase the size of proof steps, generalize the notion of an outlining construct in a natural way, refine the system based on some large case studies and explore the related issues of specification, automation, interaction. This has led to the three outlining constructs described in this paper.

Some of the other systems that have most influenced our work are HOL [GM93], Isabelle [Pau94], PVS [COR$^+$95], and Nqthm [KM96]. Many of the specification and automation techniques we utilize in Declare are derived from ideas found in the above systems. However, we do not use the proof description techniques from these systems (e.g. the HOL tactic language, or PVS strategies).

1.2 Declarative and Inferential Proof Description

For our purposes, we consider a construct to be *declarative* if it states explicitly \what" effect is achieved by the construct. Different declarations may specify different properties of a construct, e.g. type, mode and behavioral specifications in a programming language. A related question is whether the construct describes

[1] Mizar is a poorly documented system, so our observations are based on some sample Mizar scripts and the execution of the Mizar program.

\how" that effect is achieved: we will use *inferential* to describe systems that allow the omission of such details and infer them instead. Many systems are both declarative and inferential, and together they represent an ideal, where a problem statement is given in high level terms and a machine is used to infer a solution. \Inferential" is inevitably a relative notion: one system is more inferential than another if the user need specify fewer operational details. The term *procedural* is often used to describe systems that are not highly inferential, and thus typically not declarative either, i.e. systems where significant detail is needed to express solutions, and a declarative problem statement is not given.[2]

How do \declarative" and \inferential" apply to proof description? For our purposes a declarative style of proof description is one that makes the logical results of a proof step explicit:

> A proof description style is declarative if the results established by a reasoning step are evident without interpreting the justification given for those results.

Surprisingly, most existing styles of proof description are plainly *not* declarative. For example, typical HOL tactic proofs are certainly not declarative, although automation may allow them to be highly inferential. Consider the following extract from the proof of a lemma taken from Norrish's HOL formalization of the semantics of C [Nor98]:

```
val wf_type_offset = prove
  ‘‘8smap sn. well_formed_type smap (Struct sn)  !
            8fld t. lookup_field_info (smap sn) fld t  !
                   9n. offset smap sn fld n’’,
  SIMP_TAC (hol_ss ++ impnorm_set) [offset,
    definition "choltype" "lookup_field_info",
    definition "choltype" "struct_info"] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC (theorem "choltype" "well_formed_structs") THEN
  FULL_SIMP_TAC hol_ss [well_formed_type_THM] THEN
  FIRST_X_ASSUM SUBST_ALL_TAC THEN
  ...
```

Even given all the appropriate definitions, we would challenge an experienced HOL user to accurately predict the shape of the sequent late in the proof.

In an ideal world we would also like a fully inferential system, i.e. we simply state a property and the machine proves it automatically. For complex properties this is impossible, so we try to decrease the amount of information required to specify a proof. One very helpful way of estimating the amount of information contained in a proof is by looking at the dependencies inherent in it:

> One proof description style is more inferential than another if it reduces the number of dependencies inherent in the justifications for proof steps.

To give a simple concrete example, proofs in interactive theorem provers (e.g. HOL, PVS and Isabelle) are typically sensitive to the order in which subgoals

[2] Declarative and inferential ideas are, of course, very common in computing, e.g. Prolog and LaTeX are examples of languages that aspire to be both declarative and inferential.

are produced by an induction utility. That is, if the ℕ-induction utility suddenly produced the step case before the base case, then most proofs would break. There are many similar examples from existing theorem proving system, enough that proofs in these systems can be extremely fragile, or reliant on a lot of hidden, assumed detail. A major aim of proof description and applied automated reasoning must be to eliminate such dependencies where possible. Other examples of such dependencies include: reliance on the orderings of cases, variables, facts, goals and subgoals; reliance upon one of a number of logically equivalent forms of terms (e.g. $n > 1$ versus $n \geq 2$); and reliance on the under-specified behavior of proof procedures, such as how names are chosen.

2 Three Constructs for Proof Outlining

Proofs in Declare are expressed as outlines, in a language that approximates written mathematics. The constructs themselves are not radical, but our assertion is that most proof outlines can be written in these constructs and their syntactic variants alone. In other words, we assert that for many purposes these constructs are both logically and pragmatically adequate. Perhaps the most surprising thing is that large proof developments can indeed be performed in Declare even though proofs are described in a relatively restricted language.

In this section we shall describe the three primary constructs of the Declare proof language. These are:

{ First order decomposition and enrichment;
{ Proof by automation;
{ Application of second order proof principles.

2.1 Reduced Syntax

A simplified syntax of the proof language is shown below, to demonstrate how small it is at its core. We have omitted aspects that are not relevant for the purposes of this article, including specification constructs for introducing new types and constants with various properties. These include simple definitions, mutually recursive function definitions, mutually recursive fixed point specifications and algebraic datatypes. Several syntactic variations of the proof constructs are translated to the given syntax, as discussed in later sections. Declare naturally performs name resolution and type inference, the latter occurring \in context", taking into account all declarations that are visible where a term occurs. Declare also performs some syntactic reduction and comparison of terms during proof analysis, as described in Section 2.6. We have left some constructs of the language uninterpreted, in particular the language of justifications, which is discussed later in this article. Declarations also include \pragma" specifications that help indicate what various theorems mean and how they may be used by the automated reasoning engine. Finally, the terms and types are those of higher order logic as in the HOL system, extended with pattern matching as described in [Sym99].

```
Article = Decl*
Decl = thm Label "term" proof Proof end
     | ...              (other specification language constructs)
Proof = qed Justification
      | cases Justification Case* end
      | schema Label over Label
             varying Local*
          Case*
Justification = by Hint*
Case = case [Label] Enrichment* : Proof
Enrichment = [locals Local*] Fact*
Local = ident [: type]
Fact =  "term" [Label]
Label = <ident>
```

We consider a semantics describing proof checking for this language in Appendix A.

2.2 An Example

We will use a Declare proof of the following proposition to illustrate the first two constructs of the proof language: \Assume n 2 ℕ is even, and that whenever m is odd, $n=m$ is even, ignoring any remainder. Then the remainder of $n=m$ is always even." We assume that our automated reasoner is powerful enough to do some arithmetic normalization for us, e.g. collecting linear terms and distributing \mod" over + (this is done by rewriting against Declare's standard library of theorems and its built-in arithmetic procedures). We also assume the existence of the following theorems about even, odd and modulo arithmetic.

```
<even> |- even(n) = 9k. n=2*k
<odd>  |- odd(n) = 9k. n=2*k+1
<even_or_odd>  |- even(n) _ odd(n)
<div_rem_exists>  |- m > 0  !  (9d r. n=d*m+r  ^  r<m)
```

A Declare proof of this property is shown below. The constructs used and their meanings are explained in the following sections.

```
thm <mythm>
  if "m > 0"
     "odd(m)  !  even(n/m)"  <m>
     "even(n)" <n>
  then "even(n mod m)" <goal>;
proof
  consider d, r st
     "n = d*m + r"
     "r < m"  by <div_rem_exists>;

  have "d = n/m"
       "r = n mod m";
  consider n′ st
     "n = 2*n′" by <even>,<n>;

  cases by <even_or_odd> ["m"]
    case "even(m)" :
      consider m′ st "m=2*m′" by <even>;
      have "r = 2*(n′-d*m′)";
      qed by <even>,<goal>;

    case "odd(m)" :
      consider d′ st "r = 2*(n′-d′*m)" by <m>,<even>;
      qed by <even>,<goal>;
  end;
end;
```

2.3 Problem Introduction

A Declare proof begins with the statement of a problem, introduced using some variant of the `thm` declaration. The example from the previous section uses the one shown in Table 1. This variant allows us to begin our proof in a conveniently decomposed form, i.e. without outer universal quantifiers and with facts and goals already named.

External Form	Internal Form

`thm` *label* `if` *facts* `then` *goals*
`proof`
main-proof
`end`
 	
(simplified problem introduction)
	`thm` *label* "8*vars*. (⋀ *facts*) ! (⋁ *goals*)"
	`proof`
	`cases by <goal>`
	`case locals` *vs*
	facts
	goals^{-1} :
	main-proof
	`end`
	`end`
	
	where *vars* = free symbols in *facts*,*goals*
	and *goals*$^{-1}$ = *goals* with each term negated

Table 1. Syntactic variation for *Decl* with the equivalent primitive form.

2.4 Construct 1: First Order Decomposition and Enrichment

Enrichment is the process of adding facts, goals and local constants to an environment in a logically sound fashion. Most steps in vernacular proofs are enrichment steps, e.g. \consider *d* and *r* such that $n = d \; m + r$ and $r < m$." The example above illustrates how this translates into Declare's syntax. An enrichment step has a corresponding proof obligation that constants exist with the given properties. The obligation for this step is "9*d r*. n=*d**m+*r* ^ *r* < m".

This kind of enrichment is *forward reasoning*. When goals are treated as negated facts, *backward reasoning* also corresponds to enrichment. For example if our goal is 8*x*:(9*b*:*x* = 4*b*) ! even(*x*) then the vernacular \given *b* and *x* such that *x* = 4*b* then by the definition of even it suffices to show 9*c*:2 *c* = *x*" is

External Form	Internal Form
consider *vars* st *facts justification*; *main-proof* (inline introduction)	cases *justification* case locals *vars* *facts* : *main-proof* end

have *facts justification*;
main-proof
 	
(inline assertion)
	cases *justification*
	case *facts* :
	main-proof
	end

let *id* = "*term*"; *main-proof* (inline definition)	cases case locals *id* "*id* = *term*" : *main-proof* end;

sts *goal justification*;
main-proof
 	
(inline backward reasoning)
	cases *justification*
	case $goal^{-1}$: *main-proof*
	end;
	
	where $goal^{-1}$ = *goal* with the term negated

Table 2. Syntactic variations for *Proof* with equivalent primitive forms.

an enrichment step (this example is not taken from the larger example above). Based on an existing goal, we add two new local constants b and x, a new goal $9c{:}2\ c = x$ and a new fact $x = 4b$. In Declare we can use +/- to indicate new facts/goals respectively (goals are treated as negated facts), and we have:

```
consider b,x such that
   + "x = 4*b"
   - "9c. 2*c = x"
  by <goal>;        // obligation "9b x. x=4*b ^ θc. 2*c=x"
```

Decomposition is the process of splitting a proof into several cases. We combine decomposition and enrichment via the cases construct, and an example can be seen in Section 2.2. For each decomposition/enrichment there is a proof obligation that corresponds to the \default" case of the split, where we may assume each other case does not apply. Syntactically, the locals declaration for each enrichment can be omitted, as new symbols are assumed to be new local constants. The construct is very general, and some highly useful variants are translated to it as shown in Table 2, including assertion, abbreviation, and the linear forms of enrichment seen above. These forms assume the automated prover can, as a minimum, decide the trivial forms of first order equational problems that arise as proof obligations in the translations. For example, $9v{:}v = t$ is the proof obligation for the let construct, where v is not free in t.

General specification constructs could also be admitted within enrichments, e.g. to define local constants by fixed points. Declare does not implement these within proofs.

2.5 Construct 2: Appeals to Automation

At the tips of a problem decomposition we find appeals to automated reasoning to \fill in the gaps" of an argument, denoted by qed in the proof language. A set of \hints" (also called a *justification*) is provided to the engine. We shall discuss the justification language of hints in the Section 3. The automated reasoning engine is treated as an oracle, though of course the intention is that it is sound with respect to the axioms of higher order logic.

2.6 Construct 3: Second Order Schema Application

In principle, decomposition/enriching and automated proof with justifications are sufficient to describe any proof in higher order logic, assuming a modicum of power from the automated engine (e.g. that it implements the 8 primitive rules of higher order logic described by Gordon and Melham [GM93], and can decide propositional logic). However, we have found it useful to add one further construct for inductive arguments. The general form we have adopted is *second-order schema application*, which can encompass structural, rule and well-founded induction and other techniques.

Why is this construct needed? We consider a typical proposition proved by inducting over the structure of a particular set. Assume *typ list* ‘ *exp* hastype *typ* is an inductive relation defined by the four rules over a term structure as shown in Appendix B. Our example theorem states that substitution of well-typed values preserves types (we omit the definition of substitution):

```
thm <subst_safe>
if "[] ‘ v hastype xty"  <v_hastype>
   "len(E) = n"         <n>
   "(E@[xty]) ‘ e hastype ty" <typing>
then "E ‘ (subst n e v) hastype ty";
```

The induction predicate that we desire is:

```
P =  E e ty. 8n. len E = n ! E ‘ (subst n e v) hastype ty
```

One of our aims is to provide a mechanism to specify the induction predicate in a natural way. Note it is essential that *n* be universally quantified, because it is necessary to instantiate it with different values in different cases of the induction. Likewise *E*, *e* and *ty* also \vary". Furthermore, because *v* and *xty* do not vary, it is better to leave <v_hastype> out of the induction predicate to avoid extra antecedents to the induction hypothesis.

It is possible to use decomposition along with an explicit instantiation to express an inductive decomposition.

```
thm <subst_safe>
if "[] ‘ v hastype xty"  <v_hastype>
then "8E e ty.
        (E @ [xty]) ‘ e hastype ty ^
        len E = n !
          E ‘ (subst n e v) hastype ty" <goal>
proof
  let "ihyp E e ty =
        8n. len E = n ! E ‘ (subst n e v) hastype ty";
  cases by <hastype.induct> ["ihyp"], <goal>
    case + "ihyp ([dty]@(E@[xty])) bod rty" <ihyp>
         - "ihyp E (Lam dty bod) (FUN dty rty)" :
      ...
    case + "e = App f a"
         + "ihyp (E@[xty]) f (FUN dty ty)" <ihyp1>
         + "ihyp (E@[xty]) a dty" <ihyp2> :
         + "len E = n"
         - "E ‘ (subst n e v) hastype ty" :
      ...
  end;
end;
```

The induction theorem has been *explicitly instantiated*, a mechanism available in the language of justifications discussed in Section 3. Two trivial cases of the proof have been subsumed in the decomposition itself (see Section 2.8 | the cases in question correspond to the rules Int and Var in Appendix B). For the other two cases we have listed the available induction hypotheses explicitly, at two different depths of expansion | in the second case we have revealed more of the structure of the goal.

This approach is sometimes acceptable. Its advantages include flexibility, because simple cases may be omitted altogether; control, because we name the facts and constants introduced on each branch of the induction; and explicitness, which can be helpful for readability and the tool environment. Its disadvantages are an unnatural formulation of the original problem; the unnecessary repetition of induction hypotheses; a relatively complex proof obligation; and poor feedback because it is non-trivial to provide good feedback if the user makes a mistake when recording the hypotheses.

We now show how the proof appears using the schema construct of the Declare proof language.

```
thm <subst_safe>
if "[] ‘ v hastype xty"
   "len(E) = n"
    "(E@[xty]) ‘ e hastype ty" <typing>
then "E ‘ (subst n e v) hastype ty";
proof
  schema <hastype.induct> over <typing> varying n,E,ty,e
    case <Int>: ...
    case <Var>: ...
    case <Lam>
      "e = Lam dty bod"
      "ty = FUN dty rty"
      "ihyp ([dty]@(E@[xty])) bod rty"  <ihyp> :
      ...
    case <App>
      "e = App f a"
      "ihyp (E@[xty]) f (FUN dty ty)" <f_ihyp>
      "ihyp (E@[xty]) a dty"         <a_ihyp> :
```

```
      ...
  end;
end;
```

Actually, a little simpler is the `induct` variant of the `schema` construct, which chooses a default induction principle based on the predicate used to define the inductive set. The first line of the proof could have been written:

```
  induct over <typing> varying n,E,ty,e
```

Thus Declare provides one very general construct for decomposing problems along syntactic lines based on a second-order proof principle, along with some simple variants. The induction predicate is determined automatically by indicating those local constants V that "vary" during the induction. Effectively we tell Declare to reformulate the problem so some local "constants" become universally quantified, and then apply the induction principle. The induction hypothesis is thus the conjunction of all the axioms in the current logical context that contain a member of V. This gives a declarative specification of the induction predicate without contorting the initial specification of the problem.

The schema must be a fact in the logical environment of the form:

$$(\forall v_1 .\ ihyps_1 \rightarrow P\,v_1) \wedge \ldots \wedge (\forall v_n .\ ihyps_n \rightarrow P\,v_n) \rightarrow (\forall v .\ R[v] \rightarrow P\,v)$$

Equational constraints are encoded in the induction hypotheses, and the fact denoted using `over` must be an instance $R[t]$ of $R[v]$ for some t. If $\mathbb{N}$ is an inductive subset of a type for $\mathbb{Z}$, then the schema would be:

$$(\forall i .\ i=0 \rightarrow P\ i) \wedge (\forall i .\ (\exists k .\ i=k+1 \wedge P\ k) \rightarrow P\ i) \rightarrow (\forall i .\ i \in \mathbb{N} \rightarrow P\ i)$$

The `induct` form where the schema is implicit from a term or fact is most common, however the general mechanism above allows the user to prove and use new induction principles for constructs that were not explicitly defined inductively, and allows several proof principles to be declared for the same logical construct.

Each antecedent of the inductive schema generates one new branch of the proof, so no subsumption is possible. For each case:

- If no facts are given, then the actual hypotheses (i.e. those specified in the schema) are left implicit: they become "automatic" unlabelled facts used by the automated prover.
- If facts are given, they are interpreted as "purported hypotheses" and syntactically checked to ensure they correspond to the actual hypotheses (see [Sym99] for details).

The semantics of the construct are described in full in Appendix A.

2.7 Issues Relating to Second Order Schema Application

Writing out the induction predicate is time-consuming and error-prone. The macro `ihyp` can be used to stand for the induction predicate — the user does not have to define this predicate explicitly.

It is often necessary to strengthen a goal or weaken some assumptions before using induction. This can often be done simply by stating the original goal in this way, but in a nested proof we typically prove that the stronger goal is sufficient (this is usually trivial), and before we perform an induction we purge the environment of the now irrelevant original goal, to avoid unnecessary conditions being included in the induction predicate. This means adding a \discarding" construct to the proof language. Discarding facts breaks the monotonicity proof language, so to minimize its use we have chosen to make it part of the induction construct. Our case studies suggest it is only required when significant reasoning is performed before the induction step of a proof, which is rare.

A final twist on the whole proof language that comes when describing mutually recursive inductive proofs is described in [Sym99]. Essentially we need to modify the language to accommodate multiple (conjoined) goals, if the style of the proof language is to be preserved.

2.8 A Longer Example of Decomposition/Enrichment

We now look at a longer example of the use of enrichment/decomposition, to demonstrate the flexibility of this construct. The example is similar to several that arose in our case studies, but has been modified to demonstrate several points. Assume:

- The inductive relation $c \rightsquigarrow c^0$ is defined by many rules (say 40).
- c takes a particular form $(\mathsf{A}(a;b);s)$ at the current point in our proof.
- Only 8 of the rules apply when c is of this form, and of these, 5 represent \exceptional transitions" $c \rightsquigarrow (\mathsf{E}(val);s)$. The last 3 possible transitions are given by:

$$\frac{(a;s) \rightsquigarrow (v;s') \;\_\; (b;s) \rightsquigarrow (v;s')}{(\mathsf{A}(a;b);s) \rightsquigarrow (v;s')} \qquad \frac{}{(\mathsf{A}(a;b);s) \rightsquigarrow (a;s)} \qquad \frac{}{(\mathsf{A}(a;b);s) \rightsquigarrow (b;s)}$$

We are trying to prove that the predicate `cfg_ok` is an invariant of $\rightsquigarrow$:

```
type exp = A of exp * exp | E of string
thm <cfg_ok> "cfg_ok (t,s) $ match t with
                                 A(x,y) -> term_ok(s,x) ^ state_ok(s)
                               | E(str) -> state_ok(s)";
thm <cfg_ok-invariant>
if  "c ⇝ c′" <trans>
    "c = (A(a,b),s)"
    "cfg_ok c"
then "cfg_ok c′";
```

Note the proof will be trivial in the case of the exceptional transitions, since the state is unchanged. So, how do we formulate the case analysis? Do we have to write all 40 cases? Or even all 8 which apply syntactically? No - we need specify only the interesting cases, and let the automated reasoner deduce that the other cases are trivial:

```
cases by <⇝.cases> [<trans>], <cfg_ok>, <goal>
  case "c′ = (v, s′)"
       "t = a _ t = b"
       "(t,s) ---> c′" :
    rest of proof;
  case "c′ = (t, s)"
       "t = a _ t = b" :
    rest of proof;
end;
```

The hints given to the automated reasoner are explained further in Section 3. The key point is that the structure of the decomposition does *not* have to match the structure inherent in the theorems used to justify it (i.e. the structure of the rules). There must, of course, be a logical match that can be discovered by the automated engine, but the user is given a substantial amount of flexibility in how the cases are arranged. He/she can:

{ *Subsume trivial cases.* 37 of the 40 cases inherent in the definition of ⇝ can be subsumed in justification of the split.

{ *Maintain disjunctive cases.* Many interactive splitting tools would have generated two cases for the first rule shown above, by automatically splitting the disjunct. However, the proof may be basically identical for these cases, up to the choice of t.

{ *Subsume similar cases.* Structurally similar cases may be subsumed into one branch of the proof by using disjuncts, as in the second case. This is, in a sense, a form of factorization. As in arithmetic, helpful factorizations are hard for the machine to predict, but relatively easy to check.

The user can use such techniques to split the proof into chunks that are of approximately equal difficulty, or to dispose of many trivial lines of reasoning, much as in written mathematics.

3 Justifications and Automated Reasoning

Our language separates *proof outlining* from *automated reasoning.* We adopt the principle that these are separate activities and that the proof outline should be independent of complicated routines such as simplification. The link between the two is provided by *justifications.* A spectrum of justification languages is possible. For example, we might have no language at all, which would assume the automated engine can draw useful logical conclusions efficiently when given nothing but the entire logical environment. Alternatively we might have a language that spells out deductions in great detail, e.g. the forward inference rules of an LCF-like theorem prover. It may also be useful to have domain specific constructs, such as variable orderings for model checking.

Declare provides a small set of general justification constructs that were adequate for our case studies. The constructs allow the user to:

{ Highlight facts from the logical environment that are particularly relevant to the justification;
{ Specify explicit instantiations and resolutions;
{ Specify explicit case-splits;

These constructs are quite declarative and correspond to constructs found in vernacular proofs. Facts are *highlighted* in two ways:

{ By quoting their label
{ By never giving them a label in the first place, as unlabelled facts are treated as if they were highlighted in every subsequent proof step.

The exact interpretation of highlighting is determined by the automated engine, but the general idea is that highlighted facts must be used by the automated engine for the purposes of rewriting, decision procedures, first order search and so on.

\Difficult" proofs often become tractable by automation if a few *explicit instantiations* of first order theorems are given. Furthermore, this is an essential debugging technique when problems are not immediately solvable: providing instantiations usually simplifies the feedback provided by the automated reasoning engine. In a declarative proof language the instantiations are usually easy to write, because terms are parsed in-context and convenient abbreviations are often available. Formal parameters of the instantiations can be either type directed of explicitly named, and instantiations can be given in any order. For example, consider the theorem <subst_safe> from Section 2.6. When using this theorem a suitable instantiation directive may be:

```
  qed by <subst_safe> ["[]", "0", "xty"/xty];
```

We have one named and two type-directed instantiations. After processing the named instantiation five instantiable slots remain: *e*,*v*,*E*,*n* and *ty*. Types give the instantiations *E* ! [] and *n* ! 0 and the final fact:

```
‘ 8e v ty: [] ‘ v hastype xty ^ len [] = 0 ^ ([]@[xty]) ‘ e hastype ty
     ! [] ‘ (subst 0 e v) hastype ty
```

Explicit resolution is a mechanism similar in spirit to explicit instantiation. It combines instantiation and resolution and allows a fact to eliminate an appropriate unifying instance of a literal of opposite polarity in another fact. We might have:

```
  have "[] ‘ e2 hastype xty" <e2_types> by ...
  qed by <subst_safe> ["0", <e2_types>];
```

The justification on the last line gives rise to the hint:

```
‘ 8e v ty: true ^ len [] = 0 ^ ([]@[xty]) ‘ e hastype ty
     ! [] ‘ (subst 0 e v) hastype ty
```

Declare checks that there is only one possible resolutions. One problem with this mechanism is that, as it stands in Declare, unification takes no account of ground equations available in the logical context, and thus some resolutions do not succeed where we would expect them to.

Explicit case splits can be provided by *instantiating a disjunctive fact*, *rule case analysis*, or *structural case analysis*. Rule case analysis accepts a fact indicating membership of an inductive relation, and generates a fact that specifies the possible rules that might have been used to derive this fact. Structural case analysis acts on a term belonging to a free algebra (i.e. any type with an abstract datatype axiom): we generate a disjunctive fact corresponding to case analysis on the construction of the term.

4 Assessment

We now look at the properties of the proof language we have described and compare it with other methods of proof description. The language is essentially based on *decomposing* and *enriching* the logical environment. This means the environment is *monotonically increasing* along any particular branch of the proof That is, once a fact becomes available, it remains available.[3] The user manages the environment by labelling facts and goals, and by specifying meaningful names for local constants. This allows coherent reasoning within a complicated logical context.

Mechanisms for brevity are essential within declarative proofs, since a relatively large number of terms must be quoted. Declare attempts to provide mechanisms so that the user need never quote a particular term more than once with a proof. For example one difficulty is when a formula must be quoted in both a positive and a negative sense (e.g. as both a fact and an antecedent to a fact): this happens with induction hypotheses, and thus we introduced `ihyp` macros. Other mechanisms include local definitions; type checking in context; and stating problems in sequent form.

When using the proof language, the user often declares an enrichment or decomposition, giving the logical state he/she wants to reach, and only states \how to get there" in high level terms. The user does not specify the syntactic manipulations required to get there, except for some hints provided in the justification, via mechanisms we have tried to make as declarative as possible. Often the justification is simply a set of theorem names.

4.1 Comparison

Existing theorem provers with strong automation, such as Boyer-Moore [RJ79], effectively support a kind of declarative/inferential proof at the top level | the user conjectures a goal and the system tries to prove it. If the system fails, then the user adds more details to the justification and tries again. Declare extends this approach to allow declarative decompositions and lemmas in the internals of a proof, thus giving the benefits of scope and locality.

[3] There is an exception to this rule: see Section 2.6

One traditional form of proof description uses \tactics" [MRC79]. In principle tactics simply decompose a problem in a logically sound fashion. In practice tactic collections embody an interactive style of proof that proceeds by syntactic manipulation of the sequent and existing top level theorems. The user issues proof commands like \simplify the current goal", \do induction on the first universally quantified variable" or \do a case split on the second disjunctive formula in the assumptions". A major disadvantage is that the sequent quickly becomes unwieldy, and the style discourages the use of abbreviations and complex case decompositions. A potential advantage of tactics is programmability, but in reality user-defined tactics are often examples of arcane *adhoc* programming in the extreme.

Finally, many declarative systems allow access to a procedural level when necessary. One might certainly allow this in a declarative theorem proving system, e.g. via an LCF-like programmable interface. It would be good to avoid extending the proof language itself, but one could imagine adding new plug-in procedures to the automated reasoning engine and justification language via such techniques.

4.2 Pros and Cons

Some benefits of a declarative approach are:

{ *Simplicity.* Proofs are described using only a small number of simple constructs, and the obligations can be generated without knowing the behavior of a large number of tactics.

{ *Readability.* The declarative outline allows readers to skip sections they aren't interested in, but still understand what is achieved in those sections.

{ *Re-usability.* Declarative content can often be re-used in a similar setting, e.g. the same proof decomposition structure can, in principle, be used with many different automated reasoning engines.

{ *Tool Support.* An explicit characterization of the logical effect of a construct can often be exploited by tools, e.g. for error recovery and the interactive debugging environment. See [Sym98] for a description of an interactive environment for Declare, and a detailed explanation of how a declarative proof style allows proof navigation, potentially leading to more efficient proof debugging.

A declarative outline does not, of course, come for free. In particular, new facts must be stated explicitly. Procedurally, one might describe the syntactic manipulations (modus-ponens, specialization etc.) that lead to the production of those facts as theorems, and this may be more succinct in some cases. This is the primary drawback of declarative proof.

Declare proofs are not only declarative, but also highly inferential, as the automated prover is left to prove many obligations that arise. The benefits of a highly inferential system are also clear: brevity, readability, re-usability and robustness. The cost associated with an inferential system is, of course, that the computer must work out all the details that have been omitted, e.g. the syntactic manipulations required to justify a step deductively. This is costly both in terms of machine time and the complexity of the implementation.

Proofs in Declare are relatively independent of a number of factors that are traditional sources of dependency in tactic proofs. For example, Isabelle, HOL and PVS proofs frequently contain references to assumption or subgoal numbers, i.e. indexes into lists of each. The proofs are sensitive to many changes in problem specification where corresponding Declare proofs will not be. In Declare such changes will alter the proof obligations generated, but often the obligations will still be discharged by the same justifications.

To summarize, declarative theorem proving is about making the logical effect of proof steps explicit. Inferential theorem proving is about strong automated reasoning and simple justifications. These do not come for free, but in the balance we aim to achieve benefits that can only arise from a declarative/inferential approach.

References

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Baco Raton, Florida, 1995.

[GM93] M.J.C Gordon and T.F Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Har96] J. Harrison. A Mizar Mode for HOL. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 1125 of *Lecture Notes in Computer Science*, pages 203{220, Turku, Finland, August 1996. Springer Verlag.

[KM96] Matt Kaufmann and J. Strother Moore. ACL2: An industrial strength version of Nqthm. *COMPASS | Proceedings of the Annual Conference on Computer Assurance*, pages 23{34, 1996. IEEE catalog number 96CH35960.

[MRC79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[Nor98] Michael Norrish. *C Formalized in HOL*. PhD thesis, University of Cambridge, August 1998.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[RJ79] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1979.

[Rud92] P. Rudnicki. An overview of the MIZAR project, 1992. Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

[Sym98] Don Syme. Interaction for Declarative Theorem Proving, December 1998. Available from `http://research.microsoft.com/users/dsyme`.

[Sym99] Don Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, Computer Laboratory, January 1999. Available from `http://research.microsoft.com/users/dsyme`.

A A Semantics

A *logical environment* or *theory* Γ contains:

{ A signature of type and term constants;
{ A set of axioms, each of which are closed higher order logic terms.

Logical environments must always be wellformed: i.e. all their terms must typecheck with respect to their signature. Axioms are named ($label$ 7! $prop$). We can add ( ) a fragment of a logical environment to another environment. These fragments specify new types, constants and axioms. We assume the existence of a logical environment Γ_0 containing the theory of all standard propositional and rst order connectives, and other axioms of higher order logic.

The judgment $Decls$ ‘ Γ indicates that the given declarations establish Γ as a conservative extension of a minimal theory of higher order logic. The judgment Γ ‘ $Decl : \Gamma_{frag}$ is used to elaborate single declarations.

$$\frac{}{[] \text{ ‘ } \Gamma_0} \qquad \frac{\begin{array}{c} Decls \text{ ‘ } \Gamma \\ \Gamma \text{ ‘ } Decl : \Gamma_{frag} \end{array}}{Decls; Decl \text{ ‘ } \Gamma \quad \Gamma_{frag}} \qquad \frac{\begin{array}{c} prop \text{ is a closed term of type } bool \\ \Gamma \quad (\texttt{"goal"} \text{ 7! } :prop) \text{ ‘ } proof\checkmark \end{array}}{\Gamma \text{ ‘ } \texttt{thm} <lab> \; prop \; proof : (lab \text{ 7! } prop)}$$

The label "goal" is used to represent the obligation: in reality problems are specified with a derived construct in decomposed form, so this label is not used.

The relation Γ ‘ $proof\checkmark$ indicates that the proof establishes a contradiction from the information in Γ, as given by the three rules below. However rst some de nitions are required:

{ *Enriching an environment.*

$$\begin{array}{l} \Gamma \quad (\texttt{locals } c_1 ::: c_i;\ fact_1 <lab_1> ::: fact_j <lab_j>) = \\ \qquad \Gamma \quad c_1 ::: c_i \quad (lab_1 \text{ 7! } fact_1); ::: ; (lab_j \text{ 7! } fact_j) \end{array}$$

There may be no free variables in $fact_1 ::: fact_j$ besides $c_1 ::: c_i$.

{ *The obligation for an enrichment to be valid.*

$$\text{oblig}(\texttt{locals } c_1 ::: c_i; fact_1 <lab_1> ::: fact_j <lab_j>) = 9c_1 ::: c_i{:}fact_1 \ \wedge \ ::: \ \wedge \ fact_j$$

In the r.h.s, each use of a symbol c_i becomes a variable bound by the 9 quanti - cation.

{ *Discarding.* $\Gamma - labels = \Gamma$ without axioms speci ed by $labels$
{ *Factorizing.* Γ / V = the conjunction of all axioms in Γ involving any of the locals speci ed in V. When the construct is used below, each use of a local in V becomes a variable bound by the 8 quanti cation that encompasses the resulting formula.

Decomposition/Enrichment

```
proof = cases proof_0
          case lab_1 enrich_1 proof_1
          :::
          case lab_n enrich_n proof_n
        end
```

$$\frac{\begin{array}{l} \Gamma \quad (lab_1 \text{ 7! } :\text{oblig}(enrich_1); ::: ; (lab_n \text{ 7! } :\text{oblig}(enrich_n) \text{ ‘ } proof_0\checkmark \\ 8i < n{:}\ \Gamma \quad enrich_i \text{ ‘ } proof_i\checkmark \end{array}}{\Gamma \text{ ‘ } proof\checkmark}$$

Automation

$$\frac{\text{prover}(\Gamma; hints(\Gamma)) \text{ returns \textbackslash yes"}}{\Gamma \text{ ‘ qed by } hints \checkmark}$$

Schemas

```
proof = schema schema-label over fact-label
          varying V discarding discards
         case lab_1 enrich_1 : proof_1
         : : :
         case lab_n enrich_n : proof_n
       end
```

$\Gamma' = \Gamma - discards$

$\Gamma'(schema\text{-}label) = 8P{:}\,(8v{:}ihyps_1 \;\; ! \;\; P(v))$

$\quad : : :$

$\quad (8v{:}ihyps_n \;\; ! \;\; P(v))$

$\quad ! \;\; (8v{:}Q(v) \;\; ! \;\; P(v))$

$\Gamma'(fact\text{-}label) = Q(t)$

$ipred = \backslash \;\; v{:}8V{:}(\bigwedge(v = t)) \;\; ! \;\; \Gamma'{=}V''$

static matching determines that

$\quad 8v{:}\bigwedge(v = t) \;\; \wedge \;\; ihyps_i[ipred{=}P] \;\; ! \;\; \text{oblig}(enrich_i) \quad (8i{:}1 \quad i \quad n))$

$$\frac{\Gamma' \quad enrich_i \text{ ‘ } proof_i \checkmark \qquad (81 \quad i \quad n)}{\Gamma \text{ ‘ } proof \checkmark}$$

The conditions specify that:

- The proof being considered is a second-order schema application of some form;
- The given axioms are discarded from the environment (to simplify the induction predicate);
- *schema-label* specifies a schema in the current logical context of the correct form;
- *fact-label* specifies an instance of the inductive relation specified in the schema for some terms *t*. These terms may involve both locals in *V* and other constants.;
- The induction predicate is that part of the logical environment specified by the variance. If the terms *t* involve locals in the variance *V* then they become bound variables in this formula.
- Matching: the generated hypotheses must imply the purported hypotheses.
- Each sub-proof must check correctly.

B Typing Rules for the Induction Example

```
<Int>
             ----------------------------
               "E ‘ (Int i) hastype INT"

<Var>        "i < len(E) ^ ty = el(i)(E)"
            -------------------------------
               "E ‘ (Var i) hastype ty"

<Lam>       "[dty]@E ‘ bod hastype rty"
        --------------------------------------------
        "E ‘ (Lam dty bod) hastype (FUN dty rty)"

<App>    "E ‘ f hastype (FUN dty rty) ^ E ‘ a hastype dty"
        ------------------------------------------------------
                     "E ‘ (f % a) hastype rty";
```

Mechanized Operational Semantics via (Co)Induction

Simon J. Ambler and Roy L. Crole

Leicester University, Leicester LE1 7RH, U.K.

Abstract. We give a fully automated description of a small programming language *PL* in the theorem prover Isabelle98. The language syntax and semantics are encoded, and we formally verify a range of semantic properties. This is achieved via uniform (co)inductive methods. We encode notions of bisimulation and contextual equivalence. The main original contribution of this paper is a fully automated proof that *PL* bisimulation coincides with *PL* contextual equivalence.

1 Introduction

The design of new programming languages which are well-principled, reliable and expressive is an important goal of Computer Science. Semantics has a role to play here, in that it provides a firm basis for establishing the properties and behaviour of language features and programs. A number of advances in the methods of operational semantics have been made over the last few years, in particular, in the study of higher order operational semantics and program equivalences | many of these advances are detailed below. In this paper we make a contribution by considering mechanized support for reasoning about operational semantics. In particular, we specify and verify properties of a core language, using tactical verification within the theorem prover Isabelle98 [Pau94b]. We give a presentation in which the concepts are expressed as uniformly as possible using the framework for (co)inductive definitions within Isabelle-HOL. We hope that the key principles of our methodology are sufficiently flexible that they can be adapted to new languages, provided that their semantics can be given using inductive and coinductive definitions.

One framework for giving formalized semantic specifications is the structured operational semantics (SOS) introduced by Plotkin [Plo81]. This allows high level specification of programming languages, abstracting from machine level detail. It is possible to present the full semantics of a non-trivial language in this form (for example [MMH97, Cro97]). SOS allows reasoning about programs and language properties, such as the verification that program execution is deterministic or that execution respects the type discipline (for example [CG99, Gor95a, Pit97]). At a more refined level, one might want to reason about the equivalence of different programs or their termination properties. This sort of reasoning is an essential basis for program refinement techniques and for verifying compiler optimization steps. Contextual equivalence is a useful and intuitive notion of program equivalence: two programs are regarded as equivalent if they behave in the

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 221{238, 1999.

same way in all possible situations. Abramsky's applicative bisimulation gives a notion of program equivalence in the lazy -calculus [Abr90] and in fact this can be adapted to give a general notion of program equivalence. In more general situations,

{ applicative bisimulation often implies contextual equivalence (*); and
{ for *some* languages the two notions coincide.

Howe [How89] showed that bisimilarity is a congruence | this essentially means we can reason about bisimilarity using simple algebraic manipulations. The idea is that contextual equivalence is a *natural* way to compare programs | but it is difficult to establish. As noted above (*), it is often sufficient to show bisimilarity and this is frequently more tractable. So we can show two programs to be contextually equivalent by instead showing they are bisimilar. These sorts of results have been obtained for a variety of languages such as PCFL [Pit97], a functional language with IO [CG99], the Object-calculus of Abadi and Cardelli [AC96, GHL97], and the linear -calculus [Bie97, Cro96]. The papers [Las98, MT91, Pit98] discuss when variants of the \standard" theories apply.

A key step towards turning these theoretical developments to practical advantage is to produce tools which correctly embody the theory. Over the last 10 to 15 years, automated support has been developed in packages such as Isabelle and HOL [MG93]. Recently, a number of people have made advances in semantic verification. Syme has mechanized a proof of type soundness for a fragment of Java [Sym97b] using DECLARE [Sym97a]; a similar verification has been made by Nipkow and von Oheimb [NvO98] within Isabelle-HOL. Börger and Schulte [BS98] have used ASMs to formalize a Java fragment. Collins and others [CG94, MG94, Sym93] have coded fragments of Standard ML in HOL. Nipkow has verified the equivalence of alternative semantic presentations of a simple imperative language [Nip98], and Bertot and Fraer [BF96] have used COQ to verify the correctness of transformations for a related language.

There is also a body of work on the automated verification of type theory properties which are less directly concerned with the issues of programming language semantics. Altenkirch has given a proof of strong normalization of System F in the LEGO theorem prover [Alt93], and Coquand has shown normalization for simply typed lambda calculus written in ALF [Coq92]. The -calculus has been encoded in the Calculus of Constructions by Hirschkoff [Hir97].

Our contribution is to provide, within Isabelle-HOL, the operational semantics of a small programming language called *PL* together with *a fully mechanized account* of program equivalence in *PL*. In particular, we define the notions applicative bisimulation and contextual equivalence for this language and use Howe's method to prove that these coincide. This is the first time, as far as we are aware, that such a proof has been fully mechanized. Each constituent part of *PL* is encoded as a theory. Each theory gives rise to new HOL definitions, together with introduction and elimination rules. We make substantial use of the Isabelle packages for datatypes, induction and coinduction. Overall, our mechanization specifies *PL* types, expressions, a type assignment system, an operational evaluation semantics, Kleene equality of programs, a divergence predicate, contextual equivalence, and bisimilarity. We have *formally verified*:

- *determinism of evaluation,*
- *type soundness,*
- *congruence of bisimilarity,* and the
- *coincidence of bisimilarity and contextual equivalence,*

together with a number of examples of properties of particular programs. The theories produced and the dependencies between them are summarized in Figure 1.

The paper proceeds as follows. In Section 2 we give a mathematical framework for induction and coinduction. Each of our Isabelle encodings can be seen as an instance of one of the definitions from this framework. The idea is that this enables us to give a uniform account of each of the many inductive and coinductive definitions which appear in this paper. In Section 3 we describe the syntax of a small programming language *PL*, and its type system. In Section 4 we provide an operational semantics. In Section 5 we define a notion of similarity and hence bisimilarity for *PL*. In Section 6 we define a binary relation (due to Howe [How89]) which is used to facilitate proofs in Section 8. In Section 7 we encode program divergence, which is also used in Section 8. In this final section, we prove that contextual equivalence and bisimilarity do coincide.

Comments on Notation

We shall use the following conventions in this paper

- $\Longrightarrow$ denotes an Isabelle meta-level implication with single premise and conclusion .
- $[\![{}_1; \ldots ; {}_n]\!] \Longrightarrow$ denotes an Isabelle meta-level implication with multiple premises ${}_1; \ldots ; {}_n$ and conclusion .
- $\bigwedge x.$ denotes universal quantification at the Isabelle meta-level.
- $\Rightarrow$ denotes a function space in HOL.
- $\mapsto e$ denotes a function abstraction in HOL, where is a variable, and e is a HOL term.
- $\longrightarrow$ denotes an implication in HOL.
- $\forall x.$ denotes universal quantification in HOL.
- $\overset{fun}{\to}$ denotes a function space in the object language *PL*.
- (e) denotes a function abstraction in the object language *PL*, written in de Bruijn notation.

2 Inductive and Coinductive Definitions

We have achieved a uniform mechanization of *PL* by the use of inductive and coinductive methods. Recall (see, for example, [Cro98]) that if $: \mathcal{P}(X) \to \mathcal{P}(X)$

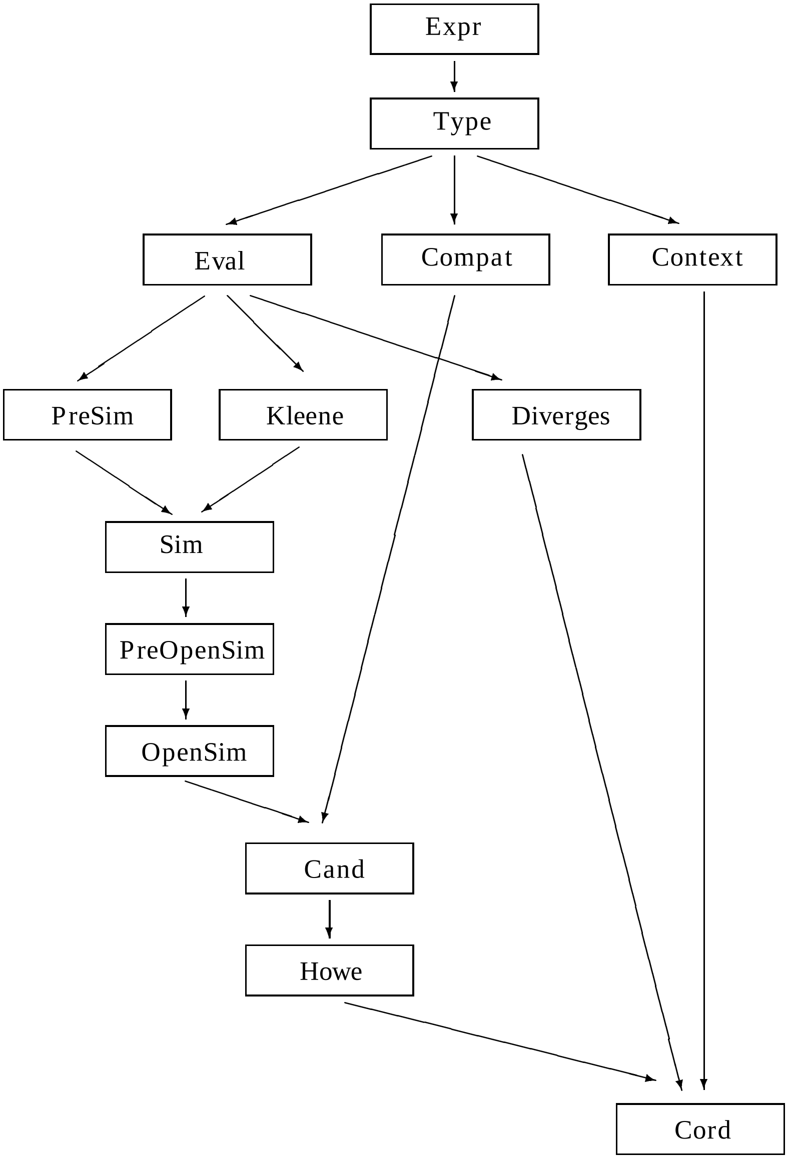


Fig. 1. Theories and dependencies between them.

is a monotone endofunction on the powerset of a set X, then the least prefixed point of is the subset of X *inductively* defined by and the greatest postfixed point is the subset of X *coinductively* defined by . The specification and properties of *PL* can be given (co)inductively. We shall show that each of these *PL* inductive and coinductive definitions can be seen as an instance of one of the following methods for defining sets and functions; and each method is itself readily coded within Isabelle using formulas ψ^{HOL} of higher order logic generated with the assistance of the (co)induction package [Pau94a]. The labels I-Set (etc) are referenced later on in the paper.

(Co)Inductive Sets and Functions

{ To $\boxed{\text{define a set } A\ 2\ P(U)}$ we specify a function $_A\colon P(U)\ !\ P(U)$ and take A to be either $_A$ (I-Set) or $_A$ (C-Set).

{ To $\boxed{\text{define a function } f\colon P(U)\ !\ P(U)}$ we specify a function $_{f;S}\colon P(U)\ !\ P(U)$ for each subset S U and take $f(S)$ to be either $_{f;S}$ (I-Function) or $_{f;S}$ (C-Function).

Throughout this paper, we show that each of semantic definitions we wish to encode in Isabelle conforms to one of the definitions of a set A or a function f given above. *Both* of these are specified by defining some other function of the form $\colon P(U)\ !\ P(U)$. However, to encode $\colon P(U)\ !\ P(U)$ in Isabelle, we simply have to define a HOL formula $\psi^{HOL}(z;S)$ such that

$$8z\colon 8S\ 2\ P(U)\colon z\ 2\ \ (S)\ \ (\ )\ \ \psi^{HOL}(z;S)$$

3 The Syntax of PL

Syntax expressions are encoded in de Bruijn notation [dB72]. The *expressions* of *PL* are given by the grammar

$$e ::= k\ j\ i\ j\ \ (e)\ j\ e\,e$$

where k ranges over constants and $i\ 2\ \mathbb{N}$ ranges over de Bruijn variable indices. The set of expressions is denoted by E. The set *Cst* of constants includes, for instance, cond, zero, suc, ncase, and rec to capture conditional tests, numbers and recursion. The theory Expr.thy defines the syntax of programs.

The *types* of *PL* are given by the grammar

$$::= nat\ j\ bool\ j\ \ \overset{fun}{!}\ \ j\ \ \ j\ list(\ )$$

The set of types is denoted by T. A *typing environment* is a list of types $[\ _0;\ldots;\ _{n-1}]$ with $_i$ indicating the type of the variable with index i in the current environment. A typical typing environment is denoted by Γ. The set

G consists of all such environments. We define a subset $Cty \quad Cst \quad T$ which supplies types for each constant, for instance,

$$(\mathsf{zero}; nat)\ 2\ Cty \qquad \text{and} \qquad (\mathsf{cond}; bool \overset{fun}{!} \quad \overset{fun}{!} \quad)\ 2\ Cty$$

for all types . The type assignment relation is generated by the usual rules of -calculus, together with an assignment $\Gamma\ `\ k\!:$ whenever $(k; \ )\ 2\ Cty$. We write $e\!:$ for $[\,]\ `\ e\!:$ and E for the set $f\ e\ j\ e\!:\ g$ of *programs* of type . The theory `Type.thy` defines the type system within Isabelle.

We have coded the syntax via -calculus using de Bruijn indices. This removes the need to axiomatize -conversion which would constitute a significant body of work before establishing any non-trivial results. Others [GM96] have shown that axiomatizing -conversion for implementation is a non-trivial task; moreover, our initial work with explicit variable names highlighted the difficulties of reasoning about the choice of fresh names for bound variables. We also investigated the use of higher order abstract syntax [JDH95] but this raised a problem in obtaining the schemes of recursion and induction over the structure of syntax needed for our proofs. Some of these issues are addressed in [JDS97] and in the recent work of Gabbay and Pitts [GP99] and Fiore, Plotkin and Turi [FPT99]. These approaches may eventually provide a viable alternative to de Bruijn encoding but, for the moment, the cost of taking a more concrete representation is well repaid in the benefit of being able to use standard schemes for dataype representation and induction. There is now a reasonable body of experience to demonstrate the effectiveness of the de Bruijn approach in proving meta-theoretic results about syntax with binding operators [Hir97, Hue94, Nip].

One cost of using de Bruijn notation is, of course, that expressions can be difficult to read. This is not a substantial problem in proving theorems about the general properties of the language as we are usually working with program expressions denoted by Isabelle variables. For work involving large explicit program fragments one might have to write a parser and pretty printer to provide a more usable interface. More significant is that de Bruijn notation replaces reasoning about the renaming of bound variables via elementary arithmetic manipulation of the variable indices. It is fortunate that this manipulation can be managed by establishing just a few well chosen lemmas because the proof of each of these can involve some quite tedious arithmetical reasoning. We next discuss the key issues of weakening and substitution.

3.1 Weakening and Substitution

The operational semantics which we have described is a substitution based system. As such, the usual properties of variable weakening and expression substitution must be encoded. This requires care. If is a type and Γ a typing environment, then $\mathtt{newVar}\ i\ \Gamma$ is the new environment formed by inserting into the ith position. The weakening of a type assignment $\Gamma\ `\ e\!:$ in *PL* requires that the indices of existing variables in e must be adjusted to take account of their new position. In Isabelle, we prove

$$[j\ \Gamma\ `\ e\!:\ ;\ j \quad \mathtt{length}(\Gamma)\ j] == )\ \mathtt{newVar}\ j\ \Gamma \quad `\ \mathtt{lift}\ e\ j\!:$$

$$\frac{}{k \Downarrow k} \qquad \frac{}{(e) \Downarrow \quad (e)} \qquad \frac{e \Downarrow \quad (e') \quad e'[e''=0] \Downarrow ve}{e\,e'' \Downarrow ve}$$

$$\frac{e \Downarrow \mathsf{cond} \quad e' \Downarrow \mathsf{true}}{e\,e' \Downarrow \quad (\quad (1))} \qquad \frac{e \Downarrow \mathsf{cond} \quad e' \Downarrow \mathsf{false}}{e\,e' \Downarrow \quad (\quad (0))} \qquad \frac{e \Downarrow \mathsf{rec} \quad e'\,(\mathsf{rec}\,e') \Downarrow ve}{e\,e' \Downarrow ve}$$

$$\frac{e \Downarrow \mathsf{suc}}{e\,e' \Downarrow \mathsf{suc}\,e'} \qquad \frac{e \Downarrow \mathsf{ncase} \quad e' \Downarrow \mathsf{zero}}{e\,e' \Downarrow \quad (\quad (1))} \qquad \frac{e \Downarrow \mathsf{ncase} \quad e' \Downarrow \mathsf{suc}\,e''}{e\,e' \Downarrow \quad (\quad (0\,e''))}$$

Fig. 2. *PL* Evaluation Relation | Function Application, Conditionals, Numbers and Recursion

where `lift` $e\ j$ is the result of incrementing each of the indices in e corresponding to a free variable beyond the jth position by one. Substitution within type assignments eliminates free variables from the current environment. In Isabelle, we prove

$$[j\ j \quad \mathtt{length}(\Gamma);\ \mathtt{newVar}\ j\ \Gamma \quad `\ e: \ ;\ \Gamma\ `\ e^0: \quad j] ==)\ \Gamma\ `\ e[e^0=j]:$$

4 Operational Evaluation

We define *operational evaluation* through judgements of the form $e + ve$ which denote a relation between expressions; the Isabelle theory is called `Eval.thy`. This binary relation is inductively defined by rules such as the examples in Figure 2. Note that we have opted for a presentation in which destructors such as cond are evaluated immediately when given a first argument. This reduces the total number of rules.

4.1 Subject Reduction

Recall that Subject Reduction is the property that the types of expressions do not change on execution. The Isabelle proof of Subject Reduction is quite elegant. We show by induction on the derivation of $e + ve$ that

$$e + ve ==)\ 8\ :e: \quad -!\ ve:$$

The inductive definition package supplied in Isabelle generates a definition in the form I-Set automatically from a set of introduction rules. The definition of a set A comes equipped with an appropriate elimination rule and induction rule which can be used on the premises of a sequent to eliminate an assumption of the form $a\ 2\ A$. In particular, the set *eval* $=$ $f(e; ve)\ j\ e + veg$ is defined from the rules in Figure 2 and corresponding induction rule allows us to eliminate an assumption of the form $e + ve$. Applying this to the goal above, we obtain a total of seventeen subgoals, each corresponding to a last step in a derivation of $e \Downarrow ve$. These are all relatively simple and can be established by the tactic `fast_tac` which searches automatically for a proof using a given selection of introduction and elimination rules and simplification rewrite rules. We supply the rules which `fast_tac` may use in the form of a *classical reasoning set*. In this case it is the default `claset()` augmented with the introduction and elimination rules for typing, together with weakening and typing substitution lemmas. The proof is completed by the following code which applies the tactic to all subgoals.

```
by (ALLGOALS (fast_tac (claset()
        addIs (typing.intrs @ typeof.intrs @ [typing_subst_0])
        addEs (typingEs @ typeofEs @ [weakening_0])
        addss simpset())));
```

Note that the use of the typing substitution lemma is in solving the subgoal

$$\bigwedge e; e'; e''; ve.\ [\![\, e \Downarrow (e');\ \forall .e : \longrightarrow (e') : ;$$
$$e'[e''/0] \Downarrow ve;\ \forall .e'[e''/0] : \longrightarrow ve : \,]\!]$$
$$\Longrightarrow \forall . e\,e'' : \longrightarrow ve :$$

which arises from the third rule of Figure 2.

This example exhibits a good style of mechanical proof. It follows closely the structure of the corresponding pen-and-paper proof and uses automation effectively to perform the case analysis via a single tactic applied uniformly. It is unfortunate, but few of our proofs were so easy. Many involved a large case analysis where the individual cases were more complex. These had to be proved one-by-one using, sometimes, rather ad-hoc methods. Here mechanical assistance gave less support.

4.2 Determinism of Evaluation

Another example of an elegant proof is that of the determinism of evaluation. We show that

$$p \Downarrow u \Longrightarrow \forall v.\ p \Downarrow v \longrightarrow v = u$$

by induction, again on the derivation of $p \Downarrow u$. The subgoals are even simpler this time. The proof in each case uses just one application of an elimination rule for *eval* and then elementary logical reasoning. The induction rule generates seventeen subgoals. The first two are trivial, but, for each of the others, applying the elimination rule for evaluation of a function application, we generate a further fifteen subgoals. There are thus a total of 227 subgoals in the proof — most are dismissed immediately!

4.3 Special Tactics

Most of the automation in the proofs has been supplied by standard search tactics such as `fast_tac`, `depth_tac` and `blast_tac` or simplification tactics such as `simp_tac`, `asm_simp_tac` and `asm_full_simp_tac`. There is one special purpose tactic which deserves mention. The rules (like those) in Figure 2 define the evaluation relation. To evaluate an expression e we apply the rules to try to prove a goal of the form $e + ?v$ where $?v$ is an uninstantiated scheme-variable. If the proof succeeds then $?v$ will become instantiated to a value for e. Unfortunately, a backwards proof search using these rules is non-deterministic, even though the evaluation strategy that they represent is deterministic. The solution is to write a special purpose tactic for evaluation based on a series of derived rules. For example, a conditional would be evaluated via the rules

$$\frac{e + ve \quad ve\, e^0 + ve^0}{e\, e^0 + ve^0} \qquad \frac{e + ve \quad \mathrm{cond}\, ve + ve^0}{\mathrm{cond}\, e + ve^0}$$

$$\frac{}{\mathrm{cond\,true} + (\quad (1))} \qquad \frac{}{\mathrm{cond\,false} + (\quad (0))}$$

Resolving against these rules in order, we ensure that the function expression is evaluated first (to the constant cond), then its argument (to either true or false), and finally the application is evaluated appropriately. The function `eval_tac` encodes this deterministic evaluation strategy for an arbitrary expression.

5 Bisimilarity for *PL*

We can now define a notion of program similarity for *PL*, coded as the Isabelle theory `Sim.thy`. Bisimilarity [Gor95b] is the symmetrization of similarity; this is a useful notion of program equivalence, as described in on page 1. This has been studied by many people (for example [CG99, Gor95a, Pit97]). Our definition of similarity is equivalent to Pitts' [Pit97], but is slightly simpler for the purposes of mechanized proof | see Section 5.2. The definition of simulation is given as a type indexed family of binary relations between closed expressions of the same type: $\preccurlyeq\ = (\preccurlyeq\ j\quad)\ 2\quad P(E\quad E\ )$. The definition of simulation is the coinductively defined set given by a certain monotone function of the form $\ :\quad P(E\quad E\ )\ !\quad P(E\quad E\ )$. Given any $R\ 2\quad P(E\quad E\ )$, one can define $\ (R)$ by specifying [Pit97] the components $\ (R)\ $ at each type $\ $; if we consider the type *bool*, our *simBool* (see below) corresponds to $\ (R)_{bool}$. The function then corresponds to our $\ _{sim}$, and $e \preccurlyeq e^0$ corresponds to our $(e; e^0)\ 2\ sim$. Let us define *simBool*, and similar sets at the remaining types.

{ *simBool* $E\quad E$ where $(e; e^0)\ 2\ simBool$ if and only if

$$(\,:9ve{:}\ e + ve)\ \_\ (e + \mathrm{true}\ \hat{}\ e^0 + \mathrm{true})\ \_\ (e + \mathrm{false}\ \hat{}\ e^0 + \mathrm{false})$$

{ *simFun*$:T\quad T\quad P(E\quad E\quad T\ )\ !\ P(E\quad E)$ where $(e; e^0)\ 2\ simFun(\ ;\ ; R)$ if and only if

$$\begin{aligned}(\,:9ve{:}\ e + ve)\ \_\ (9ve; ve^0{:}\ e + ve\ &\hat{}\ e^0 + ve^0\\ &\hat{}\ 8e_1{:}e_1 :\quad -!\ (ve\, e_1; ve^0\, e_1;\quad)\ 2\ R)\end{aligned}$$

– $simNat\colon \mathcal{P}(E \times E \times T) \to \mathcal{P}(E \times E)$ where $(e, e') \in simNat(R)$ if and only if

$$\begin{aligned}&(\neg\exists v.\, e \Downarrow v) \vee (e \Downarrow \mathsf{zero} \wedge e' \Downarrow \mathsf{zero})\\ &\quad \vee (\exists e'', e'''.\, e \Downarrow \mathsf{suc}\, e'' \wedge e' \Downarrow \mathsf{suc}\, e''' \wedge (e'', e''', nat) \in R)\end{aligned}$$

– $simPair\colon T \times T \times \mathcal{P}(E \times E \times T) \to \mathcal{P}(E \times E)$ where $(e, e') \in simPair(\sigma, \tau, R)$ if and only if

$$\begin{aligned}&(\neg\exists v.\, e \Downarrow v) \vee (\exists e_1, e_2, e'_1, e'_2.\, e \Downarrow \mathsf{pr}\, e_1\, e_2 \wedge e' \Downarrow \mathsf{pr}\, e'_1\, e'_2\\ &\qquad\qquad \wedge (e_1, e'_1, \sigma) \in R \wedge (e_2, e'_2, \tau) \in R)\end{aligned}$$

– $simList\colon T \times \mathcal{P}(E \times E \times T) \to \mathcal{P}(E \times E)$ where $(e, e') \in simList(\tau, R)$ if and only if

$$\begin{aligned}&(\neg\exists v.\, e \Downarrow v) \vee (e \Downarrow \mathsf{nil} \wedge e' \Downarrow \mathsf{nil})\\ &\quad \vee (\exists e_1, e_2, e'_1, e'_2.\, e \Downarrow \mathsf{cons}\, e_1\, e_2 \wedge e' \Downarrow \mathsf{cons}\, e'_1\, e'_2\\ &\qquad\qquad \wedge (e_1, e'_1, \tau) \in R \wedge (e_2, e'_2, list(\tau)) \in R)\end{aligned}$$

We then define the set of *simulations*, $sim \subseteq E \times E \times T$, using C-Set by taking

$$\Phi_{sim}\colon \mathcal{P}(E \times E \times T) \to \mathcal{P}(E \times E \times T)$$

where (recall page 225) $z \in \Phi_{sim}(X)$ if and only if

$$\begin{aligned}&\exists e, e'.\, z = (e, e', bool) \wedge e : bool \wedge e' : bool \wedge (e, e') \in simBool\\ &\vee \exists e, e'.\, z = (e, e', nat) \wedge e : nat \wedge e' : nat \wedge (e, e') \in simNat(X)\\ &\vee \exists e, e', \sigma, \tau.\, z = (e, e', \sigma \overset{fun}{\to} \tau)\\ &\qquad\qquad \wedge e : \sigma \overset{fun}{\to} \tau \wedge e' : \sigma \overset{fun}{\to} \tau \wedge (e, e') \in simFun(\sigma, \tau, X)\\ &\vee \exists e, e', \sigma, \tau.\, z = (e, e', \sigma \times \tau) \wedge e : \sigma \times \tau \wedge e' : \sigma \times \tau \wedge (e, e') \in simPair(\sigma, \tau, X)\\ &\vee \exists e, e', \tau.\, z = (e, e', list(\tau)) \wedge e : list(\tau) \wedge e' : list(\tau) \wedge (e, e') \in simList(\tau, X)\end{aligned}$$

This formula can be implemented in Isabelle-HOL using the (co)-induction package — see Section 5.2.

We next define open similarity [Pit97], given by judgements of the form $\Gamma \vdash e \preccurlyeq^{o} e' : \tau$. In order to define these judgements mechanically, we shall define the set $osim$ for which $(\Gamma, e, e', \tau) \in osim$ corresponds to such a judgement.

First we define the function $osimrel\colon G \times T \times T \times \mathcal{P}(G \times E \times E \times T) \to \mathcal{P}(E \times E)$ where $(e, e') \in osimrel(\Gamma, \sigma, \tau, R)$ if and only if

$$\sigma, \Gamma \vdash e : \tau \wedge \sigma, \Gamma \vdash e' : \tau \wedge \forall e_1.(e_1 : \sigma \longrightarrow (\Gamma, e[e_1/0], e'[e_1/0], \tau) \in R)$$

We can then define $osim \subseteq G \times E \times E \times T$ using C-Set. Set

$$\Phi_{osim}\colon \mathcal{P}(G \times E \times E \times T) \to \mathcal{P}(G \times E \times E \times T)$$

where $z \in \Phi_{osim}(X)$ if and only if

$$\begin{aligned}&\exists \tau, e, e'.\, z = ([\,], e, e', \tau) \wedge (e, e', \tau) \in sim\\ &\vee \exists \sigma, \tau, \Gamma, e, e'.\, z = (\mathsf{newVar}\, 0\, \Gamma\, \sigma, e, e', \tau) \in X \wedge (e, e') \in osimrel(\Gamma, \sigma, \tau, X)\end{aligned}$$

5.1 Extending Weakening and Substitution

We use Isabelle to prove variants of the syntax manipulating theorems of Section 3.1 for the *PL* judgements which take the general form $\Gamma \vdash e \; R \; e' : \tau$. For example, in the encoding of *osim*, we show that

$$[\![\, \Gamma \vdash e \preccurlyeq^{o} e' : \tau ;\; j \leq \mathtt{length}(\Gamma) \,]\!] \Longrightarrow \mathtt{newVar}\; j\; \Gamma\; \sigma \vdash \mathtt{lift}\, e\; j \preccurlyeq^{o} \mathtt{lift}\, e'\; j : \tau$$

$$[\![\, \mathtt{newVar}\; j\; \Gamma\; \sigma \vdash e \preccurlyeq^{o} e' : \tau ;\; j \leq \mathtt{length}(\Gamma);\; \Gamma \vdash e_1 : \sigma \,]\!] \Longrightarrow \Gamma \vdash e[e_1/j] \preccurlyeq^{o} e'[e_1/j] : \tau$$

These results are not trivial. The latter is established in three steps by first proving the result in the special case where Γ is empty, and then in the case where j is zero, before proving the general result. Each of these proofs proceeds by induction on the structure of the list Γ.

5.2 Mechanizing Simulations

The definition of the relations *sim* and *osim* uses auxiliary functions such as *simBool* and *simFun*. The Isabelle implementation of rule (co)induction requires us to supply a proof of the monotonicity of these functions, in order to ensure that the functions Φ_{sim} and Φ_{osim} are monotone. Originally, our definition of *sim* matched that of Pitts [Pit97] and the use of auxiliary functions was essential because of the limitations which Isabelle imposes on the form of a (co)inductive definition. The current formulation is equivalent but fits more naturally with the Isabelle implementation of rule (co)induction. The functions *simBool*, *simNat*, *simPair* and *simList* are not essential in this formulation. They can be expanded in the definition. It is only *simFun* which contains a use of universal quantification which Isabelle will not accept directly in the (co)inductive definition of a set. (We have, for completeness, automatically checked the equivalence of the two presentations.)

It is convenient to use the definition of *sim* to produce special elimination rules — such rules ensure that future proofs are well structured and greatly simplified. The elimination rule below was produced by `mk_cases`, a standard Isabelle procedure; it allows the definition of simulation at function types to be unwrapped.

$$\begin{aligned}
&[\![\, e \preccurlyeq e' : \sigma \xrightarrow{fun} \tau ;\\
&\quad \textstyle\bigwedge ve\, ve'.\, [\![\, e \Downarrow ve;\; e : \sigma \xrightarrow{fun} \tau ;\\
&\qquad\qquad e' \Downarrow ve';\; e' : \sigma \xrightarrow{fun} \tau ;\\
&\qquad\qquad \forall e_1 . e_1 : \sigma \longrightarrow ve\, e_1 \preccurlyeq ve'\, e_1 : \tau \,]\!] \Longrightarrow Q;\\
&\quad [\![\, e : \sigma \xrightarrow{fun} \tau ;\; e' : \sigma \xrightarrow{fun} \tau ;\; \forall ve''. \neg (e \Downarrow ve'') \,]\!] \Longrightarrow Q \,]\!] \Longrightarrow Q
\end{aligned}$$

There are similar rules for the other types. Well chosen elimination rules are essential for directing automatic proof procedures such as `fast_tac` and we have used the mk_cases function to generate these for most of the (co)inductive definitions in the implementation.

6 The Candidate Relation for PL

The theory Cand.thy defines an auxiliary relation between expressions with judgements of the form $\Gamma \vdash e \preccurlyeq e' :$. This relation is a precongruence, which, informally, means that we can reason about instances of the relation using simple algebraic manipulations. Further, we use Isabelle to prove that $\Gamma \vdash e \preccurlyeq e' :$ if and only if $\Gamma \vdash e \preccurlyeq^{o} e' :$ (a proof methodology was first given by [How89]). Hence $\Gamma \vdash e \preccurlyeq^{o} e' :$ is a precongruence, and thus we can also reason about similarity (and bisimilarity) using simple algebraic manipulations. The precongruence property is also used critically in other proofs.

In order to define the candidate relation mechanically, we shall define the set *cand* for which $(\Gamma; e; e'; \ ) \in$ *cand* corresponds to $\Gamma \vdash e \preccurlyeq e' :$. We first define the function

$$\mathit{compat} \colon P(G \quad E \quad E \quad T) \to P(G \quad E \quad E \quad T)$$

using **I-Function**. We define

$$_{compat;R} \colon P(G \quad E \quad E \quad T) \to P(G \quad E \quad E \quad T)$$

where $z \in {}_{compat;R}(X)$ if and only if

$$\begin{array}{l} \exists \Gamma; k; \ : z = (\Gamma; k; k; \ ) \wedge k : \\ \vee\ \exists \Gamma; e_1; e_1'; e_2; e_2'; \ ; \ : z = (\Gamma; e_1\, e_2; e_1'\, e_2'; \ ) \wedge \\ \qquad\qquad\qquad\qquad (\Gamma; e_1; e_1'; \ \overset{fun}{\to}\ ) \in R \wedge (\Gamma; e_2; e_2'; \ ) \in R \\ \vee\ \ldots\ \textit{further clauses for other types} \end{array}$$

The idea is that the clauses defining $_{compat;R}$ capture precisely the properties of a precongruence. We can use the function *compat* to define [Pit97] *cand* $G \quad E \quad E \quad T$ using **I-Set**. We define $_{cand} \colon G \quad E \quad E \quad T \to G \quad E \quad E \quad T$ where $z \in {}_{cand}(X)$ if and only if

$$\exists \Gamma; e; e'; e''; \ : z = (\Gamma; e; e''; \ ) \wedge (\Gamma; e; e'; \ ) \in \mathit{compat}(X) \wedge (\Gamma; e'; e''; \ ) \in \mathit{osim}$$

6.1 Mechanization of Candidate Order

The candidate order yields an example where proofs are more difficult to automate. Many sub-lemmas arise in proofs. Consider, for instance, the sub-lemma which states that the candidate relation is substitutive.

$$\begin{array}{l} [\![j\ \mathtt{newVar}\ j\ \Gamma \quad \vdash e_1 \preccurlyeq e_1' : \ ; j \quad \mathtt{length}(\Gamma); \Gamma \vdash e_2 \preccurlyeq e_2' : \ j]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow \Gamma \vdash e_1[e_2/j] \preccurlyeq e_1'[e_2'/j] : \end{array}$$

Gordon [Gor95b] states that the proof “follows by routine rule induction ...”. The corresponding machine proof has 77 steps. The longest example is the proof that the candidate relation is closed under evaluation,

$$[\![e \Downarrow ve;\ [\,] \vdash e \preccurlyeq e' : \tau]\!] \Longrightarrow [\,] \vdash ve \preccurlyeq e' : \tau$$

which is completed in 254 steps.

Theorem 1. *We have used Isabelle to prove mechanically that the candidate relation coincides with similarity, that is cand = osim. More informatively*

$$\Gamma \vdash e \preccurlyeq e' : \tau \longleftrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \tau$$

6.2 Comments on the Proof

That *osim* is contained in *cand* follows from the definition of *cand* and that the compatible refinement of *cand* is reflexive. The other direction is more difficult. We show that restriction of the candidate relation to closed terms is a simulation, and hence by coinduction,

$$[\,] \vdash e \preccurlyeq e' : \tau \Longrightarrow e \preccurlyeq e' : \tau$$

This follows via a series of lemmas which combine to show that *cand* has the properties required in the definition of *sim*. For example, in the case of natural numbers, we have

$$[\,] \vdash \mathsf{zero} \preccurlyeq e : \mathit{nat} \Longrightarrow e \Downarrow \mathsf{zero}$$

and

$$[\,] \vdash \mathsf{suc}\, e \preccurlyeq e' : \mathit{nat} \Longrightarrow \exists e''.\ e' \Downarrow \mathsf{suc}\, e'' \wedge [\,] \vdash e \preccurlyeq e'' : \mathit{nat}$$

These two results together with the downward closure of *cand* establish that *cand* has the requisite property of a simulation on expressions of type *nat*. The other cases follow a similar pattern. Once we have shown that the restriction *cand* is contained in *sim*, then this can be extended to open terms via

$$\Gamma \vdash e \preccurlyeq e' : \tau \Longrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \tau$$

because the relation *cand* is substitutive (as we proved in Section 6.1).

7 Divergence for *PL*

The theory `Diverges.thy` defines program divergence (looping). It formalizes the judgement $e \Uparrow$ whose meaning is that the expression e does not converge to a result. Note that divergence is also used in our proofs involving contextual equivalence. In particular, note that the clauses used in the definition of similarity in Section 5 involve judgements of the form $\neg \exists ve.\ e \Downarrow ve$. These are most easily established by showing $e \Uparrow$.

We define the set $divg \quad E$ using C-Set by taking $_{divg}\!: P(E) \;\; ! \;\; P(E)$ where $z \; 2 \quad _{divg}(X)$ if and only if

$$(9e;e^0;e^{00}\!: z = e\,e^{00} \wedge e + \quad (e^0) \wedge e^0[e^{00}\!=\!0] \; 2 \; X) \; \_ \; (9e;e^0\!: z = e\,e^0 \wedge e \; 2 \; X)$$
$\_$ . . . *further clauses for other types*

This coinductive definition of divergence can be shown to correspond to the usual definition given in terms of an infinite sequence of program transitions arising from a single-step operational semantics. We can use Isabelle to establish instances of program divergence, and this is important in reasoning about contextual equivalence.

8 Contextual Equivalence for PL

The theory `Context.thy` defines a notion of program context with judgements of the form $(\Gamma; \;\; ; \Gamma^0) \Vdash C\!: \;\;$. The intuitive idea of a context C is that it is a program module containing a parameter $-_{par}$ which indicates a position at which another program may be inserted to yield a new program. The meaning of the judgement $(\Gamma; \;\; ; \Gamma^0) \Vdash C\!: \;\;$ is that an expression e with typing $\Gamma; \Gamma^0 \;\, ` \; e\!: \;$ may be inserted into $C$ to yield an expression $(C\,e)$ with typing $\Gamma^0 \;\, ` \; (C\,e)\!: \;\;$. The theory `Cord.thy` defines a notion of contextual ordering with judgements of the form $\Gamma \;\, ` \; e \quad e^0\!: \;\;$. The intended meaning is that if the evaluation of $(C\,e)$ (denoting the context C in which e replaces the parameter) results in *ve*, then so too does the evaluation of $(C\,e^0)$. Contextual equivalence is the symmetrization of contextual ordering. We use Isabelle to prove that $\Gamma \;\, ` \; e \quad e^0\!: \;\;$ if and only if $\Gamma \;\, ` \; e \preccurlyeq^o e^0\!: \;\;$. Thus we can establish instances of contextual equivalence by instead proving bisimilarity.

We wish to define judgements of the form [CG99, Gor95a, Pit97] $\Gamma \;\, ` \; e \quad e^0\!: \;\;$, or equivalently a subset of the form $cord \quad G \quad E \quad E \quad T$. Given a set S, let $S \;\,) \;\, S$ denote the set of endofunctions on S; program contexts C are of Isabelle type $E \;\,) \;\, E$ and we omit their definition. We first define

$$ctyping \quad (G \quad T \quad G) \quad (E \;\,) \;\, E) \quad T$$

using I-Set by taking

$$_{ctyping}\!: P((G \quad T \quad G) \quad (E \;\,) \;\, E) \quad T) \;\; ! \;\; P((G \quad T \quad G) \quad (E \;\,) \;\, E) \quad T)$$

where $z \; 2 \quad _{ctyping}(X)$ if and only if

$$\begin{aligned}
&9\Gamma; \;\; \!:\!z = (([\,]; \;\; ; \Gamma); id; \;\;) \\
&\_ \; 9 \;\; ; \;\; ; \Gamma; \Gamma^0; \;\; ; e; e^0\!:\!z = ((\Gamma; \;\; ; \Gamma^0); \quad 7\!! \; (e \quad) \, e^0; \;\;) \\
&\qquad\qquad\qquad\qquad\qquad \wedge \; ((\Gamma; \;\; ; \Gamma^0); e; \quad \overset{fun}{!} \quad) \; 2 \; X \wedge \Gamma^0 \;\, ` \; e^0\!: \;\;)) \\
&\_ \; 9 \;\; ; \;\; ; \Gamma; \Gamma^0; \;\; ; e; e^0\!:\!z = ((\Gamma; \;\; ; \Gamma^0); \quad 7\!! \; e\,(e^0 \quad); \;\;) \\
&\qquad\qquad\qquad\qquad\qquad \wedge \; \Gamma^0 \;\, ` \; e\!: \quad \overset{fun}{!} \quad \wedge \; ((\Gamma; \;\; ; \Gamma^0); e^0; \;\;) \; 2 \; X
\end{aligned}$$
$\_$ . . . *further clauses for other types*

We define the *contextual ordering* to be the set *cord* $\subseteq G \times E \times E \times T$ by $(\Gamma, e, e', \sigma) \in \mathit{cord}$ if and only if

$$\Gamma \vdash e : \sigma \wedge \Gamma \vdash e' : \sigma \wedge \forall C, \tau.\, ((\Gamma, \sigma, [\,]), C, \tau) \in \mathit{ctyping} \\ \longrightarrow \exists ve.\, (C\, e) \Downarrow ve \longrightarrow \exists ve.\, (C\, e') \Downarrow ve$$

Theorem 2. *We have used Isabelle to prove that the contextual preorder coincides with similarity, that is cord = osim. More informatively,*

$$\Gamma \vdash e \sqsubseteq e' : \sigma \longleftrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \sigma$$

8.1 Comments on the Proof

The main consequence of Theorem 1 is to show that *osim* is a precongruence, since *cand* is easily proved to be precongruence. It follows that if $\Gamma, \Gamma' \vdash e \preccurlyeq^{o} e' : \sigma$ and C is any context then we can substitute e and e' into the context to get $\Gamma' \vdash (C\, e) \preccurlyeq^{o} (C\, e') : \tau$. More precisely, we have

$$[\,|\, (\Gamma, \sigma, \Gamma') \Vdash C : \tau\, ;\, \Gamma, \Gamma' \vdash e \preccurlyeq^{o} e' : \sigma \,|\,] \Longrightarrow \Gamma' \vdash (C\, e) \preccurlyeq^{o} (C\, e') : \tau$$

We deduce that *osim* is contained in *cord*. The other direction is more difficult. Much as in the proof of Theorem 1, we show that the restriction of *cord* to closed terms is a simulation, and hence by coinduction, $[\,] \vdash e \sqsubseteq e' : \sigma \Longrightarrow e \preccurlyeq e' : \sigma$. We make use of a series of lemmas to characterize the evaluation of an expression to a particular value in terms of a convergence predicate, for example, for the natural numbers,

$$e : \mathit{nat} \Longrightarrow (\exists e'.\, e \Downarrow \mathrm{suc}\, e') \longleftrightarrow \mathrm{ncase}\, e\ \bot\ (\mathrm{zero}) \Downarrow$$

The proof is quite long. There are 144 steps to establish all of the cases. Once it is done then it is not too difficult to establish that *cord* is substitutive, and so

$$\Gamma \vdash e \sqsubseteq e' : \sigma \Longrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \sigma$$

9 Conclusions and Further Research

We have successfully mechanized a core programming language, and used the encoding to verify some fundamental language properties. This has been executed with a direct and uniform methodology, much of which can be smoothly refined to deal with larger languages. We are currently undertaking a similar programme to the one described in this paper for the second order lambda calculus with fixed point recursion. Our work on *PL* has given us an infrastructure to build on and we are looking to modify and refine this to deal with the more powerful system. We will prove once again that bisimilarity and contextual equivalence coincide (though when we move to other languages, we will only be able to show that contextual equivalence is implied by bisimilarity). We are also extending this work to procedural and object oriented languages. We hope, with the benefit of experience, to develop Isabelle tactics which can be used *generically*, or at least with minor modifications, within new languages. The semantics of *PL* is presented using a SOS evaluation relation. We will also be studying other semantics including low level abstract machines. In particular, we aim to automate proofs of correspondences between high and low level semantic specifications. Each verification will increase confidence that the lower level implementation correctly encodes the higher level. The long term aim is to build a front end which will enable language designers to use such a verification system with little knowledge of Isabelle. This is crucial if such tools are to be used in real language design. We thank Andy Gordon (Microsoft) and Andy Pitts (Cambridge) who have made useful comments on our work.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65{116. Addison-Wesley, 1990.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[Alt93] T. Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 13{28. Springer-Verlag, 1993.

[BF96] Yves Bertot and Ranan Fraer. Reasoning with Executable Specifications. Technical Report 2780, INRIA, Sophia Antipolis, January 1996.

[Bie97] G. M. Bierman. Observations on a Linear PCF (Preliminary Report). Technical Report 412, University of Cambridge Computer Laboratory, 1997.

[BS98] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1998. To appear.

[CG94] G. Collins and S. Gilmore. Supporting Formal Reasoning about Standard ML. Technical Report ECS-LFCS-94-310, Laboratory for the Foundations of Computer Science, University of Edinburgh, November 1994.

[CG99] R. L. Crole and A. D. Gordon. Relating Operational and Denotational Semantics for Input/Output Effects. *Mathematical Structures in Computer Science*, 9:1{34, 1999.

[Coq92] C. Coquand. A proof normalization for simply typed lambda calculus written in ALF. In K. Petersson B. Nordström and G. Plotkin, editors, *Proc. of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[Cro96] R. L. Crole. How Linear is Howe? In G. McCusker, A. Edalat, and S. Jourdan, editors, *Advances in Theory and Formal Methods 1996*, pages 60{72. Imperial College Press, 1996.

[Cro97] R. L. Crole. The KOREL Programming Language (Preliminary Report). Technical Report 1997/43, Department of Mathematics and Computer Science, University of Leicester, 1997.

[Cro98] R. L. Crole. Lectures on [Co]Induction and [Co]Algebras. Technical Report 1998/12, Department of Mathematics and Computer Science, University of Leicester, 1998.

[dB72] N. de Bruijn. Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation, with Application to the Church Rosser Theorem. *Indagationes Mathematicae*, 34:381{391, 1972.

[FPT99] M. Fiore, G. D. Plotkin, and D. Turi. Abstract Syntax and Variable Binding. To appear in LICS 99, 1999.

[GHL97] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and Equivalence of Imperative Objects. Draft manuscript, 1997.

[GM96] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 173{190, Turku, Finland, August 1996. Springer-Verlag.

[Gor95a] A. D. Gordon. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[Gor95b] A. D. Gordon. Bisimilarity as a theory of functional programming. Technical report, Aarhus University, Denmark, 1995. BRICS Notes Series NS{ 95{3, BRICS, Aarhus University.

[GP99] M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax Involving Binders. To appear in LICS 99, 1999.

[Hir97] D. Hirschkoff. A full formalisation of -calculus theory in the calculus of constructions. In *Proceedings of TPHOL'97*, volume 1275 of *LNCS*. Springer-Verlag, 1997.

[How89] D. J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198{203, 1989.

[Hue94] G. Huet. Residual theory in -calculus: a complete Gallina development. *Journal of Functional Programming*, 4(3):371{394, 1994.

[JDH95] A. Felty J. Despeyroux and A. Hirschowitz. Higher order syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *LNCS*. Springer-Verlag, 1995.

[JDS97] F. Pfenning J. Despeyroux and C. Schüermann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, volume LNCS. Springer-Verlag, 1997.

[Las98] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Dept of Computer Science, Univ of Aarhus, 1998.

[MG93] T. F. Melham and M. J. C. Gordon. *Introduction to HOL*. Cambridge University Press, 1993.

[MG94] S. Maharaj and E. Gunter. Studying the ML module system in HOL. In T. F. Melham and J. Camilleri, editors, *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 346{361, Valletta, Malta, September 1994. Springer-Verlag.

[MMH97] R. Milner, M.Tofte, and R. Harper. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.

[MT91] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287{327, 1991.

[Nip] T. Nipkow. More Church Rosser Proofs. To appear in the Journal of Automated Reasoning.

[Nip98] T. Nipkow. Winskel is (Amost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing*, 10, 1998.

[NvO98] T. Nipkow and D. von Oheimb. Machine-Checking the Java Specification: Proving Type Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1998. To appear.

[Pau94a] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conf. on Automated Deduction*, volume 814 of *LNAI*, pages 148{161. Springer-Verlag, 1994.

[Pau94b] L.C. Paulson. Isabelle: A Generic Theorem Prover. *Lecture Notes in Computer Science*, 828, 1994.

[Pit97] A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.

[Pit98] A. M. Pitts. Existential Types: Logical Relations and Operational Equivalence. Draft paper, 1998.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI{FN 19, Department of Computer Science, University of Aarhus, Denmark, 1981.

[Sym93] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In J. J. Joyce and C.-J. H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, pages 43{60, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.

[Sym97a] D. Syme. DECLARE: A Prototype Declarative Proof System for Higher Order Logic. Technical Report 416, Computer Laboratory, University of Cambridge, February 1997.

[Sym97b] D. Syme. Proving JavaS Type Soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.

Representing WP Semantics in Isabelle/ZF

Mark Staples

University of Cambridge Computer Laboratory,
Pembroke St, Cambridge CB23QG, United Kingdom
Mark.Staples@cl.cam.ac.uk

Abstract. We present a shallow embedding of the weakest precondition semantics for a program refinement language. We use the Isabelle/ZF theorem prover for untyped set theory, and statements in our refinement language are represented as set transformers. Our representation is significant in making use of the expressiveness of Isabelle/ZF's set theory to represent states as dependently-typed functions from variable names to their values. This lets us give a uniform treatment of statements such as variable assignment, framed specification statements, local blocks, and parameterisation. ZF set theory requires set comprehensions to be explicitly bounded. This requirement propagates to the definitions of statements in our refinement language, which have operands for the state type. We reduce the syntactic burden of repeatedly writing the state type by using Isabelle's meta-logic to define a lifted set transformer language which implicitly passes the state type to statements.

Weakest precondition semantics was first used by Dijkstra in his description of a non-deterministic programming language [9]. That language was generalised to the refinement calculus by Back [3], and later by Morris [18] and Morgan [16]. Textbook presentation of weakest precondition semantics usually represent conditions on states as formulae. However, such presentations can be unclear about the logic underlying these formulae. Mechanisations of the refinement calculus must per force be precise, and two approaches have been taken. The most literal approach is to use a modal logic with program states forming the possible worlds. Work done in the Ergo theorem prover [25] is in this vein [8]. However, support for modal logics is currently unwieldy in most theorem provers. More importantly, using a modal logic hampers the re-use of results from existing classical mathematics. Another approach is to represent program states explicitly in a classical logic. Conditions can then be represented as collections of states. For example, Agerholm [2] (and later the Refinement Calculator project [28] and others [21, 13]) used predicates (functions from states to Booleans) in higher-order logic to represent sets of states.

Our approach uses this kind of representation, but we work in the untyped set theory provided by the Isabelle/ZF theorem prover. Instead of representing conditions by predicates on states, we have an essentially equivalent representation using sets of states. The main point of difference is that we use the expressiveness of untyped set theory to represent states as dependently-typed functions

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 239{254, 1999.

from variable names to values. This lets us give definitions of statements such as single and multiple variable assignment, framed specification statements, local blocks, and parameterisation.

Weakest precondition semantics is sometimes presented by inductively defining an interpretation $wp(c; q)$ over a language of refinement language statements c and arbitrary postconditions q. However, following the Refinement Calculator project [28], the mechanisation described here uses a shallow embedding [7] | statements are identified with abbreviations for their semantics. Instead of defining an interpretation operator wp, we instead define each statement as an abbreviation for a specific set transformer from a set of states representing a postcondition to a set of states representing the weakest precondition. This paper demonstrates that a shallow embedding of weakest precondition semantics can be done in a first-order setting.

The theory is intended to support the proof of refinement rules, and the subsequent development of specific program refinements. A non-trivial refinement case-study using the theory has been reported elsewhere [24]. Our use of a shallow embedding precludes the statement of some meta-results. For example, we can not state a completeness result within our theory, as there is no fixed syntax defining the limits of our language. However, if the definitions of our language constructs are appropriate, the ‘soundness’ of our refinement rules follow from our use of conservative definitions and the soundness of Isabelle/ZF.

In Sect. 1 we introduce the meta-logic of Isabelle and its object logic Isabelle/ZF. In Sect. 2 we illustrate our approach to shallow embedding by defining set transformers which do not assume any structure in the state type, and follow that in Sect. 3 by defining a lifted language of type-passing set transformers in Isabelle’s meta-logic. Section 4 demonstrates our representation of states as dependently-typed functions by defining the semantics of further statements, and Sect. 5 shows how we can modify our approach to lifting set transformers to take advantage of structure on states.

1 Isabelle: Pure and ZF

Isabelle [20] is a generic theorem prover in the LCF [10] family of theorem provers, implemented in the ML language [14]. It is generic in the sense of being a logical framework which allows the axiomatisation of various object logics. Here we briefly introduce Isabelle’s meta-logic, and the object logic Isabelle/ZF used in this work. Aspects of Isabelle not discussed here include its theory definition interface, theory management, tactics and tacticals, generic proof tools, goal-directed proof interface, advanced parsing and pretty-printing support, and growing collection of generic decision procedures.

Isabelle represents the syntax of its meta-logic and object logics in a term language which is a datatype in ML. The language provides constants, application, lambda abstraction, and bound, free and scheme variables. Scheme variables are logically equivalent to free variables, but may be instantiated during unification. These are readily utilised in the development of prototype refinement tools supporting meta-variables [19, 23]. Isabelle's meta-logic is an intuitionistic polymorphic higher-order logic. The core datatype of thm implements the axiom schemes and rules of this meta-logic. Isabelle provides constants in its meta-logic which are used to represent the rules and axioms of object logics. The meta-logical constants we make use of in this paper are:

Equality $A \equiv B$, is used to represent definitions of an object logic.

Implication is used to represent rules of an object logic, and here we write them as follows:

$$\frac{A \quad B \quad \ldots \quad C}{H}$$

Function application $F(x)$ is used to represent the application of an operator F to an operand x.

Function abstraction $\lambda x \cdot F(x)$ is used to represent the construction of parametric operators.

Isabelle also supports the definition of syntactic abbreviations, which we will write $A \rightleftharpoons B$.

Isabelle/ZF is an Isabelle object logic for untyped set theory. It is based on an axiomatisation of standard Zermelo-Fraenkel set theory in first order logic. The defined meta-level type of sets i is distinct from the meta-level type of first-order logic propositions o. These include object-level operators such as: implication $P \Rightarrow Q$; universal and existential quantification, i.e. $\forall x \cdot P(x)$ and $\exists x \cdot P(x)$; and definite description $\epsilon x \cdot P(x)$. Families of sets can be defined as meta-level functions from index sets to result sets, and operators can be defined as meta-level functions from argument sets to result propositions. From this basis, a large collection of constructs are defined and theorems about them proved. The expressiveness nature of Isabelle/ZF allows a great deal of flexibility in representing structures not allowed in simple type theory. In particular, we make use of a structure of dependent total functions $\Pi_{x:X} \cdot Y(x)$ (which we sometimes write $\Pi_X Y$) from a domain X to an X-indexed family of range types Y. The simple function space $X \to Y$ is the dependent function space where there is no dependency in Y, i.e. $\Pi_{x:X} \cdot Y$.

Other parts of the Isabelle/ZF notation which are used here include: $a : A$ or $a \in A$ for set membership, $\mathbb{P}A$ for the powerset operator, $A \subseteq B$ for subset, $\{x : A \mid P(x)\}$ for set comprehension of elements of A satisfying P, $\forall x : A \cdot P(x)$ for universal quantification, and $\bigcup A$ for the union of a set of sets. Object-logic abstraction $\lambda x : X \cdot F(x)$ is the set of pairs $\langle x, F(x) \rangle$ for all x in the explicitly declared domain X, and is distinct from meta-level lambda abstraction $\lambda x \cdot F(x)$ discussed above. Similarly, we have an infix object-logic function application $f`a$ which is distinct from the meta-level function application $F(a)$. We also make use of relational operators such as infix function overriding $f \oplus g$, domain $\mathsf{dom}(R)$, domain subtraction $R \,\mathsf{dsub}\, A$, and domain restriction $R \,\mathsf{dres}\, A$. These operators are either standard in the Isabelle/ZF distribution, or have standard definitions [24].

2 Representing Set Transformers

The refinement language we present in this section is in terms of a completely general state-type. States will come from a set A (say) and preconditions (or postconditions) will be represented by sets of states in $\mathbb{P}A$. We write $P_{A;B}$ for the function space of heterogeneous set transformers $\mathbb{P}A \rightarrow \mathbb{P}B$, and P_A for the homogeneous set transformers $P_{A;A}$. Monotonic predicate transformers play an important role in the development of theorems concerning recursion. The condition of monotonicity can be defined as follows:

$$\mathsf{monotonic}(c) \mathrel{\hat{=}} \forall a\, b : \mathsf{dom}(c).\ a \subseteq b \Rightarrow c`a \subseteq c`b$$

Note that we do not need to supply the state type as an operand to the condition of monotonicity, as we can extract it from the statement c. When $c : P_{A;B}$, then $\mathsf{dom}(c) = \mathbb{P}A$, and so we have:

$$\mathsf{monotonic}(c) = \forall a\, b : \mathbb{P}A.\ a \subseteq b \Rightarrow c`a \subseteq c`b$$

We define the set of heterogeneous monotonic set transformers as follows:

$$M_{A;B} \mathrel{\hat{=}} \{c : P_{A;B} \mid \mathsf{monotonic}(c)\}$$

We write M_A for the homogeneous monotonic set transformers $M_{A;A}$.

To illustrate our representation of set transformers, we define the skip and sequential composition statements as follows:

$$\mathsf{Skip}_A \mathrel{\hat{=}} \lambda q : \mathbb{P}A.\ q \qquad a;b \mathrel{\hat{=}} \lambda q : \mathsf{dom}(b).\ a`(b`q)$$

In the definition of the skip statement we explicitly require an operand A to represent the state type. In simple type theory this could be left implicit, but as we are using ZF, we must explicitly bound the set constructed on the right hand side of the definition. However, in the definition of the sequential composition statement we can, as in the definition of monotonicity above, extract the state type from the structure of sub-components. When $b : P_{A;B}$, we have:

$$a;b = \lambda q : \mathbb{P}A.\ a`(b`q)$$

It is easy to see that skip and sequential composition are (monotonic) set transformers. That is:

$$\mathsf{Skip}_A : M_A \qquad \frac{a : P_{B;C} \quad b : P_{A;B}}{a;b : P_{A;C}} \qquad \frac{a : M_{B;C} \quad b : M_{A;B}}{a;b : M_{A;C}}$$

The definition of skip and sequential composition illustrate our general representation scheme, and demonstrate how we can sometimes extract the state type from a statement's components. We will now show the definition of state assignment (which functionally updates the entire state), and alternation ('if-then-else') in order to illustrate the representation of expressions in our language. As it was natural to use sets of states to represent conditions on states, we similarly use sets of states to represent Boolean expressions within statements. To represent expressions of other types, we use ZF's standard representation of functions and relations as sets of pairs. State assignment and alternation are defined as follows:

$$\begin{aligned}\langle F\rangle_A &\mathrel{\widehat{=}} \lambda q{:}\mathbb{P}A.\ \{s{:}\mathrm{dom}(F) \mid F`s \in q\}\\ \text{if } g \text{ then } a \text{ else } b &\mathrel{\widehat{=}} \lambda q{:}\mathrm{dom}(a) \cup \mathrm{dom}(b).\\ &\qquad (g \cap a`q) \cup ((\bigcup(\mathrm{dom}(a) \cup \mathrm{dom}(b)) - g) \cap b`q)\end{aligned}$$

Again, we extract the state type from the structure of sub-components. However, in the definition of state assignment, we extract the state type not from a sub-statement, but instead from the expression F, which should be a state-function from some input state-type B to output state-type A. In the definition of alternation we take the union of the domains of a and b, because the final state q may be reached through either a or b. We also extract the actual underlying state-type A, using the fact that $\bigcup(\mathbb{P}A) = A$. So when $F : B \to A$ and $a, b : P_{A,B}$, we have the simpler equalities:

$$\begin{aligned}\langle F\rangle_A &= \lambda q{:}\mathbb{P}A.\ \{s{:}B \mid F`s \in q\}\\ \text{if } g \text{ then } a \text{ else } b &= \lambda q{:}\mathbb{P}A.\ (g \cap a`q) \cup ((A - g) \cap b`q)\end{aligned}$$

It is easy to see that state assignment and alternation are (monotonic) set transformers:

$$\frac{F : B \to A}{\langle F\rangle_A : M_{A,B}} \qquad \frac{a : P_A \quad b : P_A}{\text{if } g \text{ then } a \text{ else } b \ : P_A} \qquad \frac{a : M_A \quad b : M_A}{\text{if } g \text{ then } a \text{ else } b \ : M_A}$$

We have restricted alternation to be a homogeneous set transformer, but have included an extra state-type operand in the definition of state assignment, in order to make it a heterogeneous set transformer. This suits our purposes, as though we are mainly concerned with developing a 'user-level' theory of refinement laws over homogeneous set transformers, in the development of variable localisation in Sect. 4.2 we will need a heterogeneous state-assignment statement.

We can define the refinement relation in a fairly standard manner [6], but again extract the state type, so that when $a : P_{A,B}$, the definition simplifies to the following:

$$a \sqsubseteq b \iff \forall q{:}\mathbb{P}A.\ a`q \subseteq b`q$$

If $a \sqsubseteq b$, then the set of states in which we can execute b is larger than the set of states in which we can execute a. Our definition uses the domain of a for safety, so that b will have at least the behaviour guaranteed for anything which can be executed by a. We define total correctness in a standard way [4], as follows: $\{p\}\ c\ \{q\} \mathrel{\widehat{=}} p \subseteq c`q$. The refinement relation preserves total correctness. For $a, b : P_{A,B}$,

$$a \sqsubseteq b \iff \forall p{:}\mathbb{P}A\ q{:}\mathbb{P}B.\ \{p\}\ a\ \{q\} \Rightarrow \{p\}\ b\ \{q\}$$

The simplified forms of our definitions correspond fairly closely to standard definitions of statements and refinement [1, 5]. Our definitions also receive some measure of validation by the proof of theorems about our language which we would expect to be true. For example, the skip statement is the identity state-assignment, and the sequential composition of state assignments is the assignment of their compositions. That is, for $F : A \to B$ and $G : B \to C$ we have:

$$\mathsf{Skip}_A = \langle \lambda s{:}A.\, s\rangle_A \qquad \langle F\rangle_B ; \langle G\rangle_C = \langle G \circ F\rangle_C$$

These and many other standard refinement results have been proved about this set transformer in Isabelle/ZF [24]. We have now outlined our general scheme for representing set transformers, expressions, and conditions. Our purpose in this paper is to expand on this representation scheme, and so we will not labour the description of other refinement calculus statements such as magic, non-deterministic assignment, specification, demonic choice, angelic choice, and recursion. The definitions and theorems concerning these statements appear in [24]. Nonetheless, we give the definitions for statements which we use in this paper. These are the assertion, guarding and chaos statements, defined as follows:

$$\begin{aligned}
\{P\}_A &\mathrel{\hat{=}} \lambda q{:}\mathbb{P}A.\, P \cap q\\
(Q)_A \to &\mathrel{\hat{=}} \lambda q{:}\mathbb{P}A.\, (A - Q) \cup q\\
\mathsf{Chaos}_A &\mathrel{\hat{=}} \lambda q{:}\mathbb{P}A.\, \mathsf{if}(A = q, A, \emptyset)
\end{aligned}$$

These statements are well known in the refinement literature. We note that the assertion and guarding statements do not change the state, but the chaos can change the state in any manner.

We also give the definition of value-binding statements, and present a *store* statement which we will use in Sect. 4.2 to define variable-localisation statements. We call our angelic value-binding a 'logical constant' statement, after Morgan [16],[1] and dually, demonic value-binding the 'logical variable' statement. They correspond to lifted existential and universal quantification respectively, and are defined as follows:

$$\begin{aligned}
\mathsf{con}\ x.\ c(x) &\mathrel{\hat{=}} \lambda q{:}\mathsf{Dom}\, c.\ \{s : \bigcup \mathsf{Dom}\, c \mid \exists v.\, s \in c(v)`q\}\\
\mathsf{var}\ x.\ c(x) &\mathrel{\hat{=}} \lambda q{:}\mathsf{Dom}\, c.\ \{s : \bigcup \mathsf{Dom}\, c \mid \forall v.\, s \in c(v)`q\}
\end{aligned}$$

Here, the statement c is parametric upon the bound value x. We can extract the state type from c, but we must use a lifted domain operator Dom instead of the ordinary set-theoretic domain dom. Its definition makes use of the definite description operator, as follows:

$$\mathsf{Dom}\, c \mathrel{\hat{=}} \varepsilon d.\ \forall x.\, d = \mathsf{dom}(c(x))$$

[1] Our logical constant statement is different to Morgan's, as it does not bind the value of program variables. However, we use it for much the same purpose: to remember the initial value of program variables and variant functions.

We can use these value binders to define a store statement $\mathsf{store}_A\ \lambda x . c(x)$ which remembers the value x of the initial state for the execution of the sub-statement c. The store statement can be defined in terms of logical constants and assertion statements, or equivalently, in terms of logical variables and guarding statements, as follows:

$$\mathsf{store}_A\ \lambda x . c(x) = \mathsf{con}\ \lambda x . \{\{s{:}A \mid s = x\}\}_A ; c(x)$$
$$\mathsf{store}_A\ \lambda x . c(x) = \mathsf{var}\ \lambda x . (\{s{:}A \mid s = x\})_A \rightarrow ; c(x)$$

3 Lifting Set Transformers

Our language of set transformers mirrors standard definitions of statements in the refinement calculus. Unfortunately, although we can sometimes extract the state type from the domain of an operand to a statement, our use of Isabelle/ZF's set theory forces us to explicitly include the state type as an operand in most atomic statements, in some compound statements, and in every expression and condition constructed by an object-level lambda abstraction or set comprehension. We can reduce this burden by using a technique somewhat similar to the monadic style of functional programming. We define a language of lifted statements as higher-order operators in Isabelle's meta-logic. The state type is mentioned once at the top of a lifted statement, and is implicitly passed down to its sub-statements.

We define lifted skip and sequential composition statements as follows:

$$\mathsf{Skip} \mathrel{\widehat{=}} \lambda A . \mathsf{Skip}_A \qquad a ; b \mathrel{\widehat{=}} \lambda A . a(A) ; b(A)$$

We engage in a slight abuse of notation here by overloading the names of conventional and lifted statements; context will normally distinguish them. The lambda in these definitions is Isabelle's meta-logical abstraction. The lifted skip statement takes a set which will form the state type of the conventional skip statement, and the sequential composition statement takes a set which it passes to its sub-statements. We thus restrict our attention to homogeneous predicate transformers, but that suits our purposes of providing a simple 'user-level' theory of refinement.

We usually introduce the state type once at the top of an expression. Lifted equality, refinement, and monotonic predicate transformers are defined as follows:

$$a =_A b \mathrel{\widehat{=}} a(A) = b(A)$$
$$a \sqsubseteq_A b \mathrel{\widehat{=}} a(A) \sqsubseteq b(A)$$
$$a \in \mathcal{M}_A \mathrel{\widehat{=}} a(A) \in \mathcal{M}_A$$

For example, compare the lifted form $\mathsf{Skip} =_A \mathsf{Skip} ; \mathsf{Skip}$ to its equivalent set transformer statement: $\mathsf{Skip}_A = \mathsf{Skip}_A ; \mathsf{Skip}_A$. We can see that this mechanism provides a great economy of notation in the presentation of complex programs, and recovers many of the benefits of implicit typing as seen in simple type theory.

Many refinement statements have operands which we intuitively think of as expressions or conditions, but which we represent as sets. For example, we use sets of states to represent the boolean expression in the guard of the alternation statement, and we use sets of pairs of states to represent the state-valued expression in the state assignment statement. We can define 'lifted' versions of these statements which, instead of representing such operands as sets, use meta-logical predicates or functions. We can recover a set from a predicate P by using set comprehension $\{x{:}A \mid P(x)\}$, and we can recover an object-logic function as sets-of-pairs from a meta-level function F by using ZF's lambda abstraction $\lambda x{:}A.\, F(x)$. Thus our definitions of lifted alternation and state-assignment are as follows:

$$\begin{aligned} \mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b &\doteq \lambda A.\ \mathsf{if}\ \{x{:}A \mid g(x)\}\ \mathsf{then}\ a(A)\ \mathsf{else}\ b(A) \\ \langle F\rangle &\doteq \lambda A.\ \langle \lambda x{:}A.\, F(x)\rangle_A \end{aligned}$$

For example, for the following statement written using set transformers:

$$\mathsf{if}\ \{s{:}\mathbb{N} \mid s = 1\}\ \mathsf{then}\ \mathsf{Skip}_{\mathbb{N}}\ \mathsf{else}\ \langle \lambda s{:}\mathbb{N}.\, 1\rangle_{\mathbb{N}} \sqsubseteq \langle \lambda s{:}\mathbb{N}.\, 1\rangle_{\mathbb{N}}$$

we can use our language of lifted set transformers, and instead write:

$$\mathsf{if}\ \lambda s.\, s = 1\ \mathsf{then}\ \mathsf{Skip}\ \mathsf{else}\ \langle \lambda s.\, 1\rangle \sqsubseteq_{\mathbb{N}} \langle \lambda s.\, 1\rangle$$

4 Representing Program Variables

Our development so far has been in terms of a completely general state-type. Now we impose a structure on our state in order to represent program variables and their values. This allows us to define variable assignment, framed specification statements, local blocks, and parameterisation constructs. We represent states as dependent total functions of the form $\Pi_{v:V}.\, \Gamma(v)$ where V is a fixed set of variable names, and Γ is a (meta-level) variable-name indexed family of types. For variables v not under consideration in a particular development, we under-specify both the type at v and the value of v, i.e. for a state $s : \Pi_{v:V}.\, \Gamma(v)$ we would under-specify both $\Gamma(v)$ and $s`v$.

Given a typing Γ, we can recover the state type by using the abbreviation $S_\Gamma \doteq \Pi_V\, \Gamma$. We also define substitution of a new type T for a variable v, and type overriding of a new collection of variable/type pairs S as follows:

$$\begin{aligned} \Gamma[T/v] &\doteq \lambda u.\, \mathsf{if}(u = v, T, \Gamma(u)) \\ \Gamma \boxplus S &\doteq \lambda u.\, \mathsf{if}(u \in \mathsf{dom}(S), S`u, \Gamma(u)) \end{aligned}$$

4.1 Atomic Statements

Atomic statements typically use variable names in order to restrict their effect to changing the named variables. Assignment statements update the value of variables in the state functionally, whereas the choose statement and framed specification statements update the value of variables in a potentially nondeterministic way.

A single-variable assignment $v :=_A E$ changes the value of a single program variable v to the value of an expression E computed in the initial state. Multiple-variable assignment is similar, but can assign values to many variables in parallel. A multiple-variable assignment $h\vec{M}\,i_{A;B}$ contains an expression M which takes a state and returns a new set of variable/value pairs which will override variables in the initial state. We define them as follows:

$$v :=_A E \;\text{b}\; q{:}\mathbb{P}(A){:}\ fs{:}\mathrm{dom}(E)\ j\ s[^{E`s}=_v]\ 2\ qg$$
$$h\vec{M}\,i_A \;\text{b}\; q{:}\mathbb{P}(A){:}\ fs{:}\mathrm{dom}(M)\ j\ s\quad M`s\ 2\ qg$$

These two forms of variable assignment are related, i.e. when E is an expression on state-type B, we have: $v :=_A E \;=\; h\ \overrightarrow{s{:}B{:}\ fhv;E`sig}\,i_A$.

The choose statement terminates and changes its set of variables to arbitrary well-typed values while leaving other variables unchanged. It is defined as follows:

$$\mathrm{Choose}(w)_A \;\text{b}\; q{:}\mathbb{P}(A){:}\ fi{:}A\ j\ fo{:}A\ j\ i\ \mathrm{dsub}\ w = o\ \mathrm{dsub}\ wg\quad qg$$

The choose statement is like chaos on a set of variables, and we can prove that $\mathrm{Chaos}\ _{\nu} = \mathrm{Choose}(V)\ _{\nu}$. We wouldn't normally use the choose statement within our user-level theory of refinement, but we do use it in Sect. 4.2 in our definition of local blocks.

A framed specification statement $w : [P;\ \ Q]_A$ has a set of program variables w (the 'frame') which can be modified by the statement. It is defined as follows:

$$w : [P;\ \ Q]_A \;\text{b}\; q{:}\mathbb{P}(A){:}\ fi{:}A\ j\ i\ 2\ P\ \wedge$$
$$fo{:}A\ j\ o\ 2\ Q\ \wedge\ i\ \mathrm{dsub}\ w = o\ \mathrm{dsub}\ wg\quad qg$$

Framed specification can also be expressed in terms of the assertion, choose, guarding, and sequential compositions statements, i.e.:

$$w : [P;\ \ Q]_A = fPg_A;\mathrm{Choose}(w)_A;(Q)_A\ !$$

From our definitions it is possible to prove a suite of fairly standard refinement rules [24], but we do not list them here.

4.2 Compound Statements: Localisation

Compound statements typically use program variables to externally hide the effects that a sub-statement has on the named variables. For example, local blocks localise the effect of changes to declared variables, and parameterisation hides effects on formal parameters while changing actual result parameters. As is common in the refinement calculus literature [15], we separate our treatment of parameterisation from procedure calls. This paper does not deal with procedure calls, but they are discussed elsewhere [24]. We define our localisation statements (local blocks and parameterisation) in a similar way:

1. store the initial state,
2. initialise the new values for the localised variable(s),
3. execute the localised sub-statement, and
4. finally restore the original values of the localised variable(s).

These localisation statements are homogeneous set transformers which contain a differently-typed homogeneous set transformer. Initialisation and finalisation stages are therefore heterogeneous set transformers. Our development of these statements is, to some extent, based on Pratten's unmechanised formalisation of local variables [22].

Local blocks take a set of variables w and an statement c forming the body of the block. Our definition extracts the state type from c as seen before, but we must explicitly supply the external state-type A as an argument to the local block. For $c : P_B$, our definition simplifies to the following:

$$\mathrm{begin}_A\ \overline{w}'\!: c\ \mathrm{end} = \mathrm{store}_A\ s\!: \mathrm{Choose}(w)_{B;A}; c; h\overline{s\ \mathrm{dres}\ w}'i_A$$

Presentations of the refinement calculus typically consider three different kinds of parameter declaration: value, result, and value-result [17, 22]. Procedures with multiple parameters are then modelled as a composition of these individual parameterisations. However, this is not an ideal treatment: the evaluation of each parameter argument should be done in the same calling state, rather than being progressively updated by each parameter evaluation. We define a generic parameterisation statement $\mathrm{Param}_A(c; I; F)$ which represents a parameterised call to c from a calling type A. Like a multiple-variable assignment statement, parameterisation updates the value of many variables in parallel. The variables to be updated are contained in the initialisation and finalisation expressions I and F. They represent the interpretation of the actual parameters in the context of some formal parameter declaration, as discussed below in Sect. 5.2. As with local blocks, we extract the state type for the sub-statement, and supply the external (calling) type A explicitly. When $c : P_B$, the following equality holds:

$$\begin{aligned}&\mathrm{Param}_A(c; I; F) = \\ &\qquad \mathrm{store}_A\ i\!: \mathrm{Chaos}_{B;A}; h\overline{I}'(i)i_B; c; h\ \ s\!:\!B\!: i\quad F(s)i_A\end{aligned}$$

Parameterisation resembles the local block statement, except for two important differences. Firstly, for parameterisation, we use the chaos statement to help establish our initial state. This restricts us from using global variables within procedure bodies. This restriction was introduced partly to simplify reasoning about procedure calls, as a procedure body c is defined only at its declaration state-type B, but the procedure may be called from any external state-type A. Another reason for using chaos to prevent the use of global variables is that otherwise our parameterisation mechanism would model dynamic variable-binding, rather than the kind of static binding most commonly seen in imperative programming languages. A more sophisticated model of parameterisation could include an explicit list of global variables to work around these problems. Secondly, parameterisation differs from local blocks because in the finalisation part of parameterisation we don't simply restore values of localised variables. Instead, we restore the entire initial state except for variables carried in the finalisation expression F. Thus we will be able to change the value of variables given as actual parameters to result or value-result parameter declarations.

5 Lifting Revisited

In Sect. 3, we developed a lifted language of type-passing predicate transformers in order to reduce the burden of explicitly annotating set transformers with their state-types. Now that we have imposed structure on states to represent named program variables, we can modify our approach to lifting in order to take advantage of this extra structure. We also define a language of parameter declarations which we can use for our lifted parameterisation statement.

5.1 Pass Typings, Not State-Types

In our state representation, the set of variable names V is a constant. We redefine our lifted language, so that instead of passing sets in ${}_V$ representing the state-type, we will pass only the typings τ. We can reconstruct the state type S_τ from the typing τ as required. Although τ is not a set, defining a lifted language over a family of types presents no problem for Isabelle's higher-order meta-logic.

In a manner similar to our previous lifted language, we introduce typings once at the top of an expression, and implicitly pass them to any component substatements. For example, we can define lifted equality, refinement, and monotonic predicate transformers as follows:

$$\begin{aligned} a =_\tau b &\mathrel{\hat=} a(\tau) = b(\tau) \\ a \sqsubseteq_\tau b &\mathrel{\hat=} a(\tau) \sqsubseteq b(\tau) \\ a : M_\tau &\mathrel{\hat=} a(\tau) : M_{S_\tau} \end{aligned}$$

We can define the lifted skip, sequential composition, state assignment and alternation statements as follows:

$$\begin{aligned} \mathrm{Skip} &\mathrel{\hat=} \lambda\tau : \mathrm{Skip}_{S_\tau} \\ a; b &\mathrel{\hat=} \lambda\tau : a(\tau); b(\tau) \\ \langle F \rangle &\mathrel{\hat=} \lambda\tau : \langle F \rangle_{S_\tau} \\ \text{if } g \text{ then } a \text{ else } b &\mathrel{\hat=} \lambda\tau : \text{if } \{s{:}S_\tau \mid g(s)\} \text{ then } a(\tau) \text{ else } b(\tau) \end{aligned}$$

Other statements (except localisation statements, which are discussed below) can be lifted similarly.

The localisation statements discussed in Sect. 4.2 have sub-statements which act on a different state-type to the external state-type. Hence, we cannot simply pass the external typing to the sub-statement. Instead we use type declarations for localised variables to modify the typing which we implicitly pass to substatements.

For local blocks, we can declare a set of variables and their new types. We will model these declarations as a function from the local variables to their types. Hence the domain of this set will be the set of names of the local variables. We pass the external typing overwritten with the declaration types. Our definition of lifted local blocks is as follows:

$$\text{begin } D\!: c \text{ end } ƀ \quad :\text{begin}_{S_\tau}\ \overline{\text{dom}(D)}'\!: c(\quad \boxplus D) \text{ end}$$

The effect of the modified typing environment can be seen in the following refinement monotonicity theorem:

$$\frac{a : M \quad \boxplus_D \quad b : M \quad \boxplus_D \quad a \;\; v \quad \boxplus_D \;\; b}{\text{begin } D\!: a \text{ end } \;\; v \quad \text{begin } D\!: b \text{ end}}$$

When we refine a block using this theorem, the internal typing for the refinement of the body of the block is automatically calculated.

As a concrete example of the lifted syntax for blocks, consider the following unsugared refinement step, which is readily proved for $fa;ng \quad V$ with $a \notin n$, and $(a) = \mathbb{N}$:

$$
\begin{array}{ll}
 & fag : [\ \ s\!: \text{true}; \quad s\!: s`a = X \quad Y\,] \\
v & \\
 & \text{begin } fhn; \mathbb{N}ig\!: \ fa;ng : [\ \ s\!: \text{true}; \quad s\!: s`a = X \quad Y\,] \text{ end}
\end{array}
$$

Note that in this block statement the declaration is a singleton set containing a the local variable name n paired with its type. Inside the block, n has been added to the frame of the specification statement.

5.2 Lifting Parameterisation

We can lift the parameterisation statement by using a slight complication to the general scheme. A procedure body is defined at its declaration type and not at its various calling types. Instead of passing the current typing at execution, we must pass the fixed typing at declaration to the parameterised statement. For a given declaration typing S, we define the lifted parameterisation statement as follows:

$$\text{PARAM}(c; S; I; F) \ ƀ \quad :\text{Param}_{S_\tau}(c(S); I; F)$$

The $\text{PARAM}(c; S; I; F)$ statement is a general form of parameterisation. It uses initialisation and finalisation expressions embodying parameter declarations which have already been resolved. In the remainder of this section, we first describe a syntax for parameter declarations, and then show how to determine the declaration typing S and the initialisation and finalisation expressions I and F.

Formal and Actual Argument Syntax We consider three types of parameterisation common in the refinement literature: value, result, and value-result [15]. We define a collection of uninterpreted constants to use as syntactic abbreviations for declaring each kind of formal parameter declaration: $\mathsf{value}\ v : T$, $\mathsf{result}\ v : T$ and $\mathsf{valueresult}\ v : T$. The formal arguments of a procedure are represented by a list of these formal parameters.

An actual parameter is represented by the operator $\mathsf{Arg}\ a$, where a is a meta-level function representing an expression. For value parameters, this will be a normal expression, and for result and value-result parameters, this will be a constant expression returning a program variable. The actual arguments to a procedure are represented by a list of these actual parameters.

Interpreting Parameter Declarations We define three operators for interpreting lists of parameter declarations D given a typing at declaration Γ and actual argument list a: a declaration typing operator $\mathbb{T}_{D;\Gamma}$, and initialisation and finalisation operators $\mathbb{I}_{D;a}$ and $\mathbb{F}_{D;a}$. Parameterised statements c are then interpreted as $\mathsf{PARAM}(c; \mathbb{T}_{D;\Gamma}; \mathbb{I}_{D;a}; \mathbb{F}_{D;a})$.

$\mathbb{T}_{D;\Gamma}$ determines the typing internal to the declaration. Where $decl$ is any of our three forms of formal parameter declaration, we have:

$$\begin{aligned}
\mathbb{T}_{[];\Gamma} &\mathrel{\hat{=}} \Gamma\\
\mathbb{T}_{(decl\ v:T)::l;\Gamma} &\mathrel{\hat{=}} \mathbb{T}_{l;\Gamma[T=v]}
\end{aligned}$$

$\mathbb{I}_{D;a}$ determines the initialisation expression. It takes the state s outside the procedure call, so that actual arguments can be evaluated. Value parameters evaluate an expression which is substituted for the formal parameter, and value-result parameters take the value of variable given as the actual argument and substitute it for the formal parameter:

$$\begin{aligned}
\mathbb{I}_{[];[]} &\mathrel{\hat{=}} \lambda s.\emptyset\\
\mathbb{I}_{(\mathsf{value}\ v:T)::dl;(\mathsf{Arg}\ a)::al} &\mathrel{\hat{=}} \lambda s.\{\langle v, a(s)\rangle\} \cup \mathbb{I}_{dl;al}(s)\\
\mathbb{I}_{(\mathsf{result}\ v:T)::dl;(\mathsf{Arg}\ a)::al} &\mathrel{\hat{=}} \mathbb{I}_{dl;al}\\
\mathbb{I}_{(\mathsf{valueresult}\ v:T)::dl;(\mathsf{Arg}\ a)::al} &\mathrel{\hat{=}} \lambda s.\{\langle v, s`a(s)\rangle\} \cup \mathbb{I}_{dl;al}(s)
\end{aligned}$$

$\mathbb{F}_{D;a}$ determines the finalisation expression. It takes a state s outside the procedure call, so that actual variable arguments can be updated. Result and value-result parameters update the value of the variable given as the actual argument with the value of the formal parameter:

$$\begin{aligned}
\mathbb{F}_{[];[]} &\mathrel{\hat{=}} \lambda s.\emptyset\\
\mathbb{F}_{(\mathsf{value}\ v:Ty)::dl;(\mathsf{Arg}\ a)::al} &\mathrel{\hat{=}} \mathbb{F}_{dl;al}\\
\mathbb{F}_{(\mathsf{result}\ v:Ty)::dl;(\mathsf{Arg}\ a)::al} &\mathrel{\hat{=}} \lambda s.\{\langle a(s), s`v\rangle\} \cup \mathbb{F}_{dl;al}(s)\\
\mathbb{F}_{(\mathsf{valueresult}\ v:Ty)::dl;(\mathsf{Arg}\ a)::al} &\mathrel{\hat{=}} \lambda s.\{\langle a(s), s`v\rangle\} \cup \mathbb{F}_{dl;al}(s)
\end{aligned}$$

For example, assume we have a simple procedure declared (at a typing) as follows:

$$\begin{array}{l}\text{procedure } P([\text{value } a : A;\ \text{result } b : B;\ \text{valueresult } c : C])\ ƀ \\ Body\end{array}$$

Here a, b and c are variables in V. Now, consider the following procedure call:

$$P([\text{Arg}\ \ s{:}\ \ (s);\ \text{Arg}\ \ s{:}\ \ ;\ \text{Arg}\ \ s{:}y])$$

Here is an expression of type A, and , and y are program variables in V, with s‘ $: B$ and s‘$y : C$. This procedure call can be expanded to the following parameterised statement:

$$\text{PARAM}(Body;\ \ [C{=}c][B{=}b][A{=}a];\ \ s{:}ha;\ \ (s)i;hc;s‘yi;\ \ s{:}h\ \ ;s‘bi;hy;s‘ci)$$

Note that the value parameters a and c are updated in the initialisation expression, and the result arguments and y are updated in the finalisation expression. In an actual program refinement this expanded form of procedure call need not be seen by the developer, as it is built into the definition of procedure declaration. Our treatment of procedure and recursive procedure declarations is described further elsewhere [24].

6 Conclusions

Refinement languages are more difficult to formalise than ordinary programming languages. During program refinement, a developer may want to introduce a local variable (whether initialised or not) whose values will come from some new abstract type. In advance of performing a refinement we may not know which types we will use, and so our type universe can't be fixed ahead of time.

For statements acting on completely general state-types, our approach and definitions follow Agerholm's mechanisation for program verification [2] and the Refinement Calculator project [28], both in the HOL theorem prover [11]. Their representation of states uses a polymorphic product of values, where 'variable names' are projection functions from the tuple. So, they do not have a type of all variable names. A limitation of their approach is that procedure call rules must be re-proved for every procedure at each of their calling state-types [27]. The approach outlined in this paper does not suffer from this problem.

We make effective use of the expressiveness of Isabelle/ZF's set theory to model states, but ZF set theory brings with it the disadvantage that we must explicitly bound set comprehensions and lambda abstractions. This means that state types must appear as operands in our definition of set transformers. In order to mitigate against this syntactic overhead, we have been able to use Isabelle's higher-order meta-logic to abstract the state type (or in Sect. 5, the typing). This kind of technique may be able to be used in other settings. For example, explicitly typed type theories may, in a similar way, be able to abbreviate fragments of their theory corresponding to simple type theory.

Kleymann [12], in the context of machine-checked proofs of soundness and completeness results for program verification, reports a representation of states similar to ours. Instead of using an untyped set theory, Kleymann uses the rich type theory provided by the Lego proof tool. Pratten [22] and von Wright [26] also use an approach somewhat similar to ours. However, they use a constant global variable typing which does not, for example, change within the scope of a local block. Our statements depend on a state type operand, so we can change the state type under consideration when we enter a local block, or when we call a parameterised procedure.

We represent states as total functions from variables, and have not as yet investigated the use of partial dependently-typed functions. This might facilitate reasoning about the limits of variable sharing for the parallel composition of statements.

Acknowledgments

Comments from anonymous referees have helped to improve this paper.

References

[1] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] S. Agerholm. Mechanizing program verification in HOL. Master's thesis, Computer Science Department, Aarhus University, April 1992.

[3] R. J. R. Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Abo Akademi University, 1978.

[4] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593{624, 1988.

[5] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 42{66. Springer-Verlag, 1989.

[6] R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2(3):247{272, 1990.

[7] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, and John Herbert. Experience with embedding hardware description languages. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A10 of *IFIP Transactions*, pages 129{156. North-Holland/Elsevier, June 1992.

[8] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Refinement in Ergo. Technical Report 94-44, Software Verification Research Centre, The University of Queensland, July 1995.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[12] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, Laboratory for Foundations of Computer Science, The University of Edinburgh, September 1998.

[13] J. Knappmann. A PVS based tool for developing programs in the refinement calculus. Master's thesis, Christian-Albrechts-University of Kiel, October 1996.

[14] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[15] Carroll Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17{28, 1988.

[16] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403{419, July 1988.

[17] Carroll Morgan. *Programming from Specifications*. Prentice-Hall International, 2nd edition, 1994.

[18] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287{306, December 1987.

[19] R. G. Nickson and L. J. Groves. Metavariables and conditional refinements in the refinement calculus. Technical Report 93-12, Software Verification Research Centre, The University of Queensland, 1993.

[20] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[21] H. Pfeifer, A. Dold, F. W. von Henke, and H. Rueß. Mechanised semantics of simple imperative programming constructs. Technical Report UIB-96-11, Universität Ulm, December 1996.

[22] Chris H. Pratten. *Refinement in a Language with Procedures and Modules*. PhD thesis, Department of Engineering and Computer Science, University of Southampton, June 1996.

[23] Mark Staples. Window inference in Isabelle. In L. Paulson, editor, *Proceedings of the First Isabelle User's Workshop*, volume 379, pages 191{205. University of Cambridge Computer Laboratory Technical Report, September 1995.

[24] Mark Staples. *A Mechanised Theory of Refinement*. PhD thesis, Computer Laboratory, University of Cambridge, November 1998. Submitted.

[25] M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, The University of Queensland, February 1994.

[26] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31, 1994.

[27] J. von Wright. Verifying modular programs in HOL. Technical Report 324, University of Cambridge Computer Laboratory, January 1994.

[28] J. von Wright. *A Mechanised Calculus of Refinement in HOL*, January 27, 1994.

A HOL Conversion for Translating Linear Time Temporal Logic to ! -Automata[?]

Klaus Schneider and Dirk W. Hoffmann

University of Karlsruhe, Department of Computer Science,
Institute for Computer Design and Fault Tolerance (Prof. D. Schmid),
P.O. Box 6980, 76128 Karlsruhe, Germany,
{Klaus.Schneider, Dirk.Hoffmann}@informatik.uni-karlsruhe.de,
http://goethe.ira.uka.de/

Abstract We present an embedding of linear time temporal logic LTL in HOL together with an elegant translation of LTL formulas into equivalent ! -automata. The translation is completely implemented by HOL rules and is therefore safe. Its implementation is mainly based on preproven theorems such that the conversion works very efficiently. In particular, it runs in linear time in terms of the given formula. The main application of this conversion is the sound integration of symbolic model checkers as (unsafe) decision procedures in the HOL theorem prover. On the other hand, the conversion also enables HOL users to directly verify temporal properties by means of HOL's induction rules.

1 Introduction

Specifications of reactive systems such as digital hardware circuits are conveniently given in temporal logics (see [1] for an survey). As the system that is to be checked can be directly viewed as a model of temporal logic formulas, the verification of these specification is usually done with so-called model checking procedures which have found a growing interest in the past decade. Tools such as SMV [2], SPIN [3], COSPAN [4], HSIS [5], and VIS [6] have already found bugs in real-world examples [7, 8, 9] with more than 10^{30} states.

While this is an enormous number for control-oriented systems, this number of states is quickly exceeded if data paths are involved. In these cases, the verification with model checking tools often suffers from the so-called state-explosion problem which roughly means that the number of states grows exponentially with the size of the implementation.

For this reason, we need interactive theorem provers such as HOL [10] for the verification of systems that are build up from control and data paths. However, if lemmata or specifications about the control flow are to be verified which do not affect the manipulation of data, then the use of a model checker is often more convenient, whereas the HOL proofs are time-consuming and tedious. This is even more true, if the verification fails and the model checker is able to present a counterexample. Therefore, it is a broadly

[?] This work has been financed by DFG project 'Verification of embedded systems' and the ESPRIT LTR Project 26241 (Prosper).

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 255–272, 1999.

accepted claim that neither the exclusive use of model checkers nor the exclusive use of theorem provers is sufficient for an efficient verification of entire systems.

Instead, we need a combination of tools of both categories. For this reason, the integration of model checkers as (unsafe) tactics of the theorem prover are desirable. We use the notion ‘unsafe’ to indicate that the validity of the theorem has been checked by an external tool that is trusted, i.e., the validity is *not* checked by HOL’s rules.

The HOL theorem prover has already some built-in decision procedures as e.g. `ARITH_CONV`, but a decision procedure for temporal logics, and even more an embedding of temporal logic at all is currently missing, although there is a long-term interest of the HOL users in temporal logic. In fact, one main application of the HOL system is the verification of concurrent systems (cf. the aim of the Prosper project http://www.dcs.gla.ac.uk/prosper). Other theorem provers for higher order logic such as PVS have already integrated model checkers [11].

For a sound integration of a model checker in the HOL theorem prover, we have to address the problem that we must first embed a temporal logic in the HOL logic. The effort of such an embedding depends crucially on the kind of temporal logic that is to be embedded. In general there are linear time and branching time temporal logics. Linear time temporal logics LTL state facts on computation paths of the systems, i.e., subformulas of this logic are viewed as HOL formulas of type $\mathbb{N}$! $\mathbb{B}$ (where $\mathbb{N}$ represents the type of natural numbers and $\mathbb{B}$ the type of Boolean values). Branching time temporal logics, as, e.g., CTL* [12] can moreover quantify over computation paths, i.e., they can express facts as ‘for all computation paths leaving this state some linear time property holds’ or ‘there is a computation path leaving this state that satisfies some linear time property’.

We will see in the following that the embedding of LTL is straightforwardly done in HOL since we only have to define the temporal operators as functions that are applied on arguments of type $\mathbb{N}$! $\mathbb{B}$. Embeddings of branching time temporal logics would additionally require to formalize the computation paths of the system under consideration and therefore cause much more effort. This is the reason why we have decided to embed LTL in HOL instead of the more powerful logic CTL*.

Unfortunately the most efficient model checking tools, namely symbolic model checkers such as SMV, are not able to directly deal with LTL specifications. Instead, they are only able to handle a very restricted subset of CTL* called CTL [13] which is a subset of the alternation free -calculus. For this reason, model checking can be reduced to the computation of fixpoints.

Hence, the use of symbolic model checking procedures as decision procedures for LTL formulas in HOL requires the conversion of the LTL formulas into a format that the model checker can handle. In general, it is not possible to translate LTL to CTL, although there are some fragments of LTL that allow this [14]. However, it is well-known that LTL can be translated to finite-state !-automata so that verifying an LTL formula is reduced to a language containment problem. The latter can be presented as a fixpoint problem that can be finally solved with a symbolic model checker.

Therefore, the contribution of this paper is twofold: first, we present an embedding of LTL together with a large theory of preproven theorems that can be used for various purposes. The verification of specifications given in this temporal logic can be verified with traditional means in HOL and is then not limited to propositional temporal logic. For example it also allows the verification of properties as the ones given in [15]. Second, we have implemented a transformation of LTL into a generalized form of non-deterministic Büchi automata in the form of a HOL conversion. It is to be noted that the implementation only makes use of primitive HOL rules and is therefore save (i.e., the `mk_thm` function is not used). Using this conversion, we can translate each LTL formula into an equivalent !-automaton and have then the choice to verify the remaining property by means of traditional HOL tactics or by calling a model checker such as SMV. An implementation of the theory and the conversion is freely available under http://goethe.ira.uka.de/~schneider/.

The paper has the following outline: in the next section, we list some related work done in the HOL community and discuss then several approaches for the translation of LTL into finite state !-automata that have been developed outside HOL. In Section 4, we present our embedding of LTL and the description of !-automata in HOL. Section 5 describes the algorithm for transforming LTL formulas into generalized Büchi automata and Section 6 explains the application of the work within the Prosper project[1]. The paper then ends with some conclusions.

2 Embedding Temporal Logic and Automata in HOL

Pioneering work on embedding automata theory in HOL has been done by Loewenstein [16, 17]. Schneider, Kumar and Kropf [18] presented non-standard proof procedures for the verification of finite state machines in HOL. A HOL theory for finite state automata for the embedding of hardware designs has been presented by Eisenbiegler and Kumar [19]. Schneider and Kropf [20] presented a unified approach based on automaton-like formulas for combining different formalisms. In this paper, we have, however, no real need for an automata theory and have therefore taken a direct representation of automata by means of HOL formulas similar to [20] as given in section 4.2.

In the domain of temporal logics, Agerholm and Schjodt [21] were the first who made a model checker available for HOL. Von Wright [22] mechanized TLA (Temporal Logic of Actions) [23] in HOL. Andersen and Petersen [24] have implemented a package for defining minimal or maximal fixpoints of Boolean function transformers. As applications of their work, they embedded the temporal logic CTL [13] and Unity [25] in HOL which enables them to reason about Unity programs in HOL [26, 27] (which was their primary aim).

However, none of the mentioned authors considered the temporal logic LTL, although this logic is often preferred in specification since it is known to be a very readable temporal logic. Therefore, the work that comes closest to the one presented here is probably done by Schneider [28] (also [20]) who presented a subset of LTL that can be translated to deterministic !-automata by means of closures. This paper presents however another approach to translate full LTL to !-automata, but can be combined with the approach of [28] to reduce the number of states of the automaton [29].

[1] http://www.dcs.gla.ac.uk/prosper

3 Translating Linear Time Temporal Logic to !-Automata

In this section, we discuss the state of the art of the translation of LTL formulas to !-automata. Introductions to temporal logics and !-automata are given in [1] and [30], respectively.

Translations of LTL to !-automata has been studied in a lot of papers. Among the first ones is the work of Lichtenstein and Pnueli [31, 32] and those of Wolper [33, 34]. These procedures explicitly construct an !-automaton whose states are sets of subformulas of the given LTL formula. Hence, the number of states of the obtained automaton is of order $2^{O(n)}$ where n is the length of the considered LTL formula. Hence, also the translation procedure for converting a given LTL formula to an equivalent !-automaton is of order $2^{O(n)}$.

Another approach for translating LTL into !-automata has been given in [35]. There, a deterministic Muller automaton is generated by a bottom-up traversal through the syntax tree of the LTL formula. Boolean connectives are reduced to closures under complementation, union and intersection of the corresponding automata. The closure under temporal operators is done by first constructing a nondeterministic automaton that is made deterministic afterwards. Clearly, this approach suffers from the high complexity of the determinization algorithm which has to be applied for each temporal operator. Note that the determinization of !-automata is much more complex than the determinization of finite-state automata on finite words: while for any nondeterministic automaton on finite words with n states there is an equivalent one with $2^{O(n)}$ states, it has been proved in [36] that the optimal bound for !-automata is $2^{O(n\log(n))}$.

An elegant way for translating LTL into !-automata by means of *alternating !-automata* has been presented in [37]. Similar to the method presented in this paper, the translation with alternating !-automata runs in linear runtime and space (in terms of the length of the given LTL formula), but the resulting alternating !-automata cannot be easily handled with standard model checkers. Therefore, if we used this conversion, we would need another translation from alternating !-automata to traditional !-automata with an exponential blow-up.

The translation procedure we use neither needs to determinize the automata nor is it based on alternating !-automata. Its runtime is nevertheless very efficient: we can translate any LTL formula of length n in time $O(n)$, i.e., in linear runtime in an !-automaton with $2^{O(n)}$ states. The trick is not to construct the automaton explicitly, since this would clearly imply exponential runtime.

The translation goes back to [38] where it has been used for the implementation of a LTL front-end for the SMV model checker. In [39] it is shown that this translation can even be generalized to eliminate linear time operators in arbitrary temporal logic specifications so that even a model checking procedure for CTL* can be obtained.

4 Representing !-Automata and Temporal Logic in HOL

The translation procedure that we use is a special case of the product model checking procedure given in [39] and has been presented in [38]. We first give the essential idea of the translation procedure and its implementation as a HOL conversion and give then the details of the LTL embedding and the formal presentation of the translation procedure.

The first step of the translation procedure is the computation of a ‘definitional normal form’ for a given LTL formula , which looks in general as given below:

$$9‘_1 ::: ‘_n . \overset{n}{\underset{i=1}{}} \quad ‘_i = '_i \quad \overset{!}{} \wedge \Psi$$

In the above formula, $'_i$ contains exactly one temporal operator which is the top-level operator of $'_i$. Moreover, $'_i$ contains only the variables that occur in plus the variables $‘_1,\ldots, ‘_{i-1}$. Ψ is a propositional formula that contains the variables $‘_1,\ldots, ‘_n$ and the variables occurring in , so that could be obtained from Ψ by replacing the variables $‘_1,\ldots, ‘_n$ with $'_1,\ldots, '_n$, respectively, in this order.

The computation of this normal form is essentially done by the function tableau that is given in Figure 2. The equivalence between the above normal form and the original formula can be proved by a very simple HOL tactic. One direction is proved by stripping away all quantifiers and the Boolean connectives so that a rewrite step with the assumptions proves the goal. The other direction is simply obtained by instantiating the witnesses $'_i$, respectively. In HOL syntax, the tactic is written as follow:

```
EQ_TAC THENL
   [REPEAT STRIP_TAC;
     MAP_EVERY EXISTS_TAC [ '1; : : : ; 'n]]
THEN ASM_REWRITE_TAC[]
```

Having computed this normal form and proven the equivalence with our original formula , we then make use of preproven theorems that allow us to replace the ‘definitions’ $‘_i = '_i$ by equivalent nondeterministic ! -automata[2]. Repeating this substitution, we finally end up with a product ! -automaton, whose initial states are determined by Ψ, and whose transition relation and acceptance condition is formed by the equations $‘_i = '_i$.

4.1 Representing Linear Time Temporal Logic in HOL

We now present the formal definition of the linear time temporal logic that is considered in this paper. After defining syntax and semantics of the logic, we also present how we have embedded the logic in a very simple manner in the HOL logic. So, let us first consider the definition of the linear time temporal logic LTL.

Definition 1 (Syntax of LTL). The following mutually recursive definitions introduce the set of formulas of LTL over a given set of variables V :

- each variable in V is a formula, i.e., V LTL
- $: '$, $' \wedge$, $' \_$ 2 LTL if $';$ 2 LTL
- X $'$, [$'$ $\underline{\mathsf{U}}$] 2 LTL if $';$ 2 LTL

[2] See http://goethe.ira.uka.de/~schneider/my_tools/ltlprover/Temporal_Logic.html.

Informally, the semantics is given relative to a considered point of time t_0 as follows: $\mathsf{X}a$ holds at time t_0 iff a holds at time $t_0 + 1$; $[a\ \underline{\mathsf{U}}\ b]$ holds at time t_0 iff there is a point of time $t + t_0$ in the future of t_0 where b becomes true and a holds until that point of time.

For a formal presentation of the semantics, we first have to define models of the temporal logic: Given a set of variables V_Σ, a model for LTL is a function π of type $\mathbb{N} \to \wp(V_\Sigma)$ where $\wp(M)$ denotes the powerset of a set M. The interpretation of formulas is then done according to the following definition.

Definition 2 (Semantics of LTL). Given a finite a set of variables V_Σ and a model $\pi : \mathbb{N} \to \wp(V_\Sigma)$. Then, the following rules define the semantics of LTL:

- $\pi \models x$ iff $x \in \pi^{(0)}$
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$
- $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $\pi \models \psi$
- $\pi \models \mathsf{X}\varphi$ iff $\lambda t.\pi^{(t+1)} \models \varphi$
- $\pi \models [\varphi\ \underline{\mathsf{U}}\ \psi]$ iff there is a $\delta \in \mathbb{N}$ such that $\lambda t.\pi^{(t+\delta)} \models \psi$ and for all $d < \delta$ it holds that $\lambda t.\pi^{(t+d)} \models \varphi \wedge \neg\psi$

As a generalization to the above definition, temporal logics such as LTL are often interpreted over finite state Kripke structures. These are finite state transition systems where each state is labeled with a subset of variables of V_Σ. To interpret a LTL formula along a path π of such a Kripke structure, the above definition is used. As it is however easily possible to define a set of admissible paths in the form of a finite state transition system (similar to our representation of finite-state ω-automata as given below), we can ‘mimic’ Kripke structures easily in HOL without the burden of deep-embedding Kripke structures.

It is also easily seen that we can consider the projections $\pi_x := \lambda t.\pi^{(t)} \cap \{x\}$ for all variables $x \in V_\Sigma$ instead of the path $\pi : \mathbb{N} \to \wp(V_\Sigma)$ itself. Using these projections, we can reestablish the path π as $\pi := \lambda t.\bigcup_{x \in V_\Sigma} \pi_x^{(t)}$. Hence, it is only a matter of taste whether we use the path π or its projections π_x as a model for the LTL logic. Using the projections directly leads to our HOL representation of LTL formulas: We represent variables of LTL in the HOL logic directly as HOL variables of type $\mathbb{N} \to \mathbb{B}$. So this simplification of the semantics allows a very easy treatment of LTL in HOL that even circumvents a deep-embedding of the LTL syntax at all. With this point of view, we have the following definitions of temporal operators, i.e., the embedding of LTL, in HOL:

Definition 3 (Defining Temporal Operators in HOL). The definition of temporal operators X and $\underline{\mathsf{U}}$ in HOL is as follows (for any $p, q : \mathbb{N} \to \mathbb{B}$):

- $\mathsf{X}p := \lambda t.p^{(t+1)}$
- $[p\ \underline{\mathsf{U}}\ q] := \lambda t.\exists\delta.q^{(t+\delta)} \wedge \forall d.d < \delta \to p^{(t+d)} \wedge \neg q^{(t+d)}$

Note that X is of type $(\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})$ and $\underline{\mathsf{U}}$ is of type $(\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})$.

Clearly, it is possible and desirable to have more temporal operators that describe other temporal relationships as the ones given above. We found that the following set of temporal operators turned out to be adequate for practical use and have therefore added these operators to our LTL theory:

$$\begin{array}{ll}
\mathsf{G}p = & t{:}8d{:}p^{(t+d)} \\
\mathsf{F}p = & t{:}9d{:}p^{(t+d)}_{\mathrm{h}} \qquad\qquad \mathrm{i} \\
[p\ \mathsf{U}\ q] = & t{:}\ \ [\mathsf{F}p]^{(t)}\ \ !\ \ [p\ \underline{\mathsf{U}}\ q]^{(t)}\ \ \wedge\ \ {:}[\mathsf{F}p]^{(t)}\ \ !\ \ \mathsf{G}p^{(t)} \\
[p\ \underline{\mathsf{B}}\ q] = & t{:}9\ \ {:}p^{(t+d)}\wedge\ \ 8d{:}d\qquad !\ \ {:}q^{(t+d)} \\
[p\ \mathsf{B}\ q] = & t{:}8\ {:}\ \ 8d{:}d<\qquad !\ \ {:}q^{(t+d)}\ \ \wedge q^{(t+\ \ )}\ \ !\ \ \ 9d{:}d<\ \ \ \wedge p^{(t+d)} \\
[p\ \underline{\mathsf{W}}\ q] = & t{:}9\ \ {:}p^{(t+d)}\wedge q^{(t+d)}\ \ 8d{:}d<\qquad !\ \ {:}q^{(t+d)} \\
[p\ \mathsf{W}\ q] = & t{:}8\ {:}\ \ 8d{:}d<\qquad !\ \ {:}q^{(t+d)}\ \ \wedge q^{(t+d)}\ \ !\ \ p^{(t+d)}
\end{array}$$

G, F, and [U] are the usual always, eventually, and until operators, respectively. There are some remarkable properties that can be found in our HOL theory on temporal operators. For example, we can compute a negation normal form NNF(’) of a formula ’ by the following equations:

$$\begin{array}{ll}
{:}[\mathsf{G}p]^{(t)} = [\mathsf{F}[\ \ t{:}\,{:}\,p^{(t)}]]^{(t)} & {:}\ [p\ \mathsf{W}\ q]^{(t)} = \ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \underline{\mathsf{W}}\ p\ \ ^{(t)} \\
{:}[\mathsf{F}p]^{(t)} = [\mathsf{G}[\ \ t{:}\,{:}\,p^{(t)}]]^{(t)} & {:}\ [p\ \underline{\mathsf{W}}\ q]^{(t)} = \ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \mathsf{W}\ p\ \ ^{(t)} \\
{:}\ [p\ \underline{\mathsf{U}}\ q]^{(t)} = \ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \mathsf{B}\ p\ \ ^{(t)} & {:}\ [p\ \mathsf{B}\ q]^{(t)} = \ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \underline{\mathsf{U}}\ p\ \ ^{(t)} \\
{:}\ [p\ \mathsf{U}\ q]^{(t)} = \ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \underline{\mathsf{B}}\ p\ \ ^{(t)} & {:}\ [p\ \underline{\mathsf{B}}\ q]^{(t)} = \ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \mathsf{U}\ p\ \ ^{(t)}
\end{array}$$

Also, it is shown that any of the binary temporal operators can be expressed by any other binary temporal operator. Hence, we could restrict our considerations, e.g., to the temporal operators X and U̲ and use the following reduction rules:

$$\begin{array}{l}
\mathsf{G}p = \ \ t{:}\,{:}\ \ [\ \ t{:}\mathsf{T}]\ \underline{\mathsf{U}}\ [\ \ t{:}\,{:}\,p^{(t)}]\ \ ^{(t)} \\
\mathsf{F}p = [p\ \underline{\mathsf{U}}\ [\ \ t{:}\mathsf{T}]] \\
[p\ \mathsf{U}\ q] = \ \ t{:}\,[p\ \underline{\mathsf{U}}\ q]^{(t)}\ \_\ [\mathsf{G}p]^{(t)} \\
[p\ \mathsf{B}\ q] = \ \ t{:}\,{:}\ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \underline{\mathsf{U}}\ q\ \ ^{(t)} \\
[p\ \mathsf{W}\ q] = \ \ t{:}\ \ [\ \ t{:}\,{:}\,q^{(t)}]\ \underline{\mathsf{U}}\ [\ \ t{:}p^{(t)}\wedge q^{(t)}]\ \ ^{(t)}\ \_\ [\mathsf{G}[\ \ t{:}\,{:}\,q^{(t)}]]^{(t)} \\
[p\ \underline{\mathsf{W}}\ q] = \ \ t{:}\ \ [\ \ t{:}\,{:}\,q^{(t)}]\ \underline{\mathsf{U}}\ [\ \ t{:}p^{(t)}\wedge q^{(t)}]\ \ ^{(t)} \\
[p\ \underline{\mathsf{B}}\ q] = \ \ t{:}\,{:}\ \ [\ \ t{:}\,{:}\,p^{(t)}]\ \underline{\mathsf{U}}\ q\ \ ^{(t)}\wedge [\mathsf{F}p]^{(t)}
\end{array}$$

Note, however, that the computation of the negation normal NNF(’) of a formula ’ form as given above reintroduces the B operator, so that we deal with X, U̲, and B in the following.

4.2 Representing ! -Automata in HOL

Let us now consider how we represent ! -automata in HOL. In general, an ! -automaton consists of a finite state transition system where transitions between two states are enabled if a certain input is read. A given sequence of inputs then induces one or more sequences of states that are called *runs over the input word*. A word is accepted iff there is a run for that word satisfying the acceptance condition of the automaton. Different kinds of acceptance conditions have been investigated, consider e.g. [30] for an overview.

The representation of an $!$-automaton as a formula in HOL is straightforward: We encode the states of the automaton by a subset of $\mathbb{B}^n$. Hence, a run is encoded by a finite number of state variables $q_0; : : : ; q_n$ which are all of type $\mathbb{N} \ ! \ \mathbb{B}$. Similarly, we encode the input alphabet by a subset of $\mathbb{B}^m$ (or isomorphic $\}(fx_0; : : : ; x_m g)$) and input sequences with variables $x_0; : : : ; x_m$ of type $\mathbb{N} \ ! \ \mathbb{B}$. Then, we represent an $!$-automaton as a HOL formula of the following form:

$$
\begin{array}{l}
9q_0 : : : q_n: \\
\quad \text{h} \ _I(q_0^{(0)}; : : : ; q_n^{(0)}) \wedge \\
\quad\; 8t: \ _R(q_0^{(t)}; : : : ; q_n^{(t)}; x_0^{(t)}; : : : ; x_m^{(t)}; q_0^{(t+1)}; : : : ; q_n^{(t+t)}) \ \text{i} \ \wedge \\
\quad\; _F(q_0; : : : ; q_n)
\end{array}
$$

$_I(q_0^{(0)}; : : : ; q_n^{(0)})$ is thereby a propositional formula where only the atomic formulas $q_0^{(0)}; : : : ; q_n^{(0)}$ may occur. $_I$ represents the set of initial states of the automaton. Any valuation of the atomic formulas $q_0^{(0)}; : : : ; q_n^{(0)}$ that satisfies $_I$ is an initial state of the automaton. Hence, the set of initial states is the set of Boolean tuples $(b_0; : : : ; b_n) \ 2 \ \mathbb{B}^n$ such that $_I(b_0; : : : ; b_n)$ is equivalent to T.

Similarly, $_R(: : :)$ is a propositional formula where only the atomic formulas $q_i^{(t)}$, $x_i^{(t)}$, and $q_i^{(t+1)}$ may occur. $_R$ represents the transition relation of the $!$-automaton as follows: there is a transition from state $(b_0; : : : ; b_n) \ 2 \ \mathbb{B}^n$ to the state $(b_0^0; : : : ; b_n^0) \ 2 \ \mathbb{B}^n$ for the input $(a_0; : : : ; a_m) \ 2 \ \mathbb{B}^m$, iff $_R(b_0; : : : ; b_n; a_0; : : : ; a_m; b_0^0; : : : ; b_n^0)$ is equivalent to T.

$_F(q_0; : : : ; q_n)$ is the acceptance condition of the automaton. Note that $_R$ may be partially defined, i.e., there may be input sequences x_i that have no run through the transition system, i.e., the formula may not be satisfied even without considering the acceptance condition. In general, the following types of acceptance conditions are distinguished, where all formulas $_k$, Ψ_k are propositional formulas over $q_0^{(t_1+t_2)}; : : : ; q_n^{(t_1+t_2)}$:

Büchi:	$8t_1 :9t_2 : \ _0$
Generalized Büchi:	$\text{V}_{k=1}^{a} [8t_1 :9t_2 : \ _k]$
Streett:	$\text{V}_{k=1}^{a} [8t_1 :9t_2 : \ _k] \ \_ \ [9t_1 :8t_2 :\Psi_k]$
Rabin:	$\text{W}_{k=1}^{a} [8t_1 :9t_2 : \ _k] \wedge [9t_1 :8t_2 :\Psi_k]$

It can be shown that the nondeterministic versions of the above $!$-automata have the same expressive power [30]. Therefore, we could use any of them for our translation. In the following, we focus on generalized Büchi automata. It will become clear after the next section, why this kind of $!$-automaton is an appropriate means for a simple translation of temporal logic inside HOL.

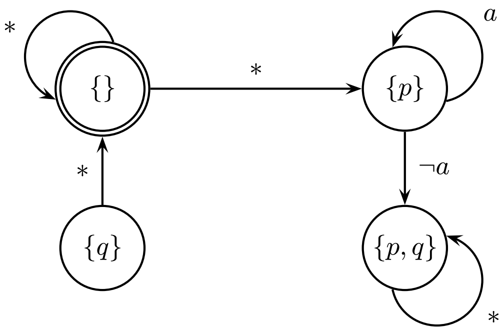


Fig.1. An example ! -automaton

As an example of the representation of an ! -automaton, consider Figure 1. This ! -automaton is represented by the following HOL formula:

$$
\begin{array}{l}
9p : \mathbb{N} \ ! \ \mathbb{B}: 9q : \mathbb{N} \ ! \ \mathbb{B}: \\
\quad :p^{(0)} \wedge :q^{(0)} \\
\quad \wedge 8t: \begin{array}{l} (p^{(t)} \ ! \ p^{(t+1)}) \wedge (p^{(t+1)} \ ! \ p^{(t)} \_ :q^{(t)}) \wedge \\ (q^{(t+1)} = (p^{(t)} \wedge :q^{(t)} \wedge :a^{(t)}) \_ (p^{(t)} \wedge q^{(t)})) \end{array} \\
\quad \wedge 8t_1:9t_2:p^{(t_1+t_2)} \wedge :q^{(t_1+t_2)}
\end{array}
$$

This means that the only initial state is the one where neither p nor q does hold (drawn with double lines in figure 1). The transition relation is simple as given in figure 1. The acceptance condition requires that a run must visit infinitely often the state where p holds, but q does not hold. Hence, any accepting run starts in state fg and must finally loop in state fpg so that the automaton formula is equivalent to $9t_1:8t_2:a^{(t_1+t_2)}$.

5 The Tableau Procedure

Several tableau procedures have been designed for various kinds of logics, e.g., first order logic, -calculi, and, of course, temporal logics. While the standard tableau procedure for LTL has been presented in [33, 34], we follow the procedure given in [38]. This works roughly as follows: for each subformula ’ that starts with a temporal operator and contains no further temporal operators, a new state variable ‘, : ℕ ! 𝔹 is generated. On each path, the state transitions of the automaton should be such that the variable ‘, exactly behaves like the subformula ’ does, hence ‘, = ’ must hold along each path. By applying the substitution of subformulas recursively, we can reduce a given LTL formula to a set of equations $E = f‘_i = ’_i \ j \ i \ 2 \ Ig$ and some propositional formula Ψ such that $\bigvee_{i2I} \mathsf{G}[‘_i = ’_i] \ ! \ (\ = \Psi)$ is valid. Clearly, if we resubstitute the variables $‘_i$ by $’_i$ in Ψ, we obtain the original formula Ψ again (syntactically).

In the following, we simplify the syntax to obtain more readable formulas: we neglect applications on time and -abstraction of the time variable. For example, we write $: [[:p] \, \underline{\mathsf{U}} \, q] \wedge [\mathsf{F}p]$ instead of $t:: \ [\ t::p^{(t)}] \, \underline{\mathsf{U}} \, q \ ^{(t)} \wedge [\mathsf{F}p]^{(t)}$. It is clear from the context, where applications on time and -abstractions should be added to satisfy the type rules.

To illustrate the computation of the ‘definitional normal form’, consider the formula (FGa) ! (GFa). The construction of E and Ψ results in E = f‘$_1$ = Ga; ‘$_2$ = F‘$_1$; ‘$_3$ = Fa; ‘$_4$ = G‘$_3$g and Ψ = ‘$_2$! ‘$_4$. The essential step is now to construct a transition relation out of E for an !-automaton with the state variables ‘$_i$ such that each ‘$_i$ behaves as ’$_i$. This is done by the characterization of temporal operators as fixpoints as given in the next theorem.

Theorem 1 (Characterizing Temporal Operators as Fixpoints). *The following formulas are valid, i.e., they hold on each path:*

G[y = a ^ Xy] = G[y = Ga] _ G[y = F]
G[y = a _ Xy] = G[y = Fa] _ G[y = T]
G[y = (b) aj Xy)] = G[y = [a W b]] _ G[y = [a <u>W</u> b]]
G[y = b _ a ^ Xy] = G[y = [a U b]] _ G[y = [a <u>U</u> b]]
G[y = :b ^ (a _ Xy)] = G[y = [a B b]] _ G[y = [a <u>B</u> b]]

Hence, each of the equations y = a ^ Xy, y = a _ Xy, y = (b) aj Xy), y = b _ a ^ Xy, *and* y = :b ^ (a _ Xy), *has exactly two solutions for* y.

The following formulas are also valid and show how one of the two solutions of the above fixpoint equations can be selected with different fairness constraints:

G[y = Ga] = G[y = a ^ Xy] ^ GF[a ! y]
G[y = Fa] = G[y = a _ Xy] ^ GF[y ! a]
G[y = [a W b]] = G[y = (b) aj Xy)] ^ GF[y _ b]
G[y = [a <u>W</u> b]] = G[y = (b) aj Xy)] ^ GF[y ! b]
G[y = [a U b]] = G[y = b _ a ^ Xy] ^ GF[y _ :a _ b]
G[y = [a <u>U</u> b]] = G[y = b _ a ^ Xy] ^ GF[:y _ :a _ b]
G[y = [a B b]] = G[y = :b ^ (a _ Xy)] ^ GF[y _ a _ b]
G[y = [a <u>B</u> b]] = G[y = :b ^ (a _ Xy)] ^ GF[:y _ a _ b]

If we define an ordering relation on terms of type ℕ ! 𝔹 by : , 8t: $^{(t)}$! $^{(t)}$, then we can also state that Ga is the greatest fixpoint of f$_a$(y) := a ^ Xy, and so on. Consequently, the above theorem characterizes each temporal operator as a least or greatest fixpoint of some function.

As can be seen, the strong and weak binary temporal operators satisfy the same fixpoint equations, i.e., they are both solutions of the same fixpoint equation. Furthermore, the equations of the first part show that there are exactly two solutions of the fixpoint equations. The strong version of a binary operator is the least fixpoint of the equations, while the weak version is the greatest fixpoint. Hence, replacing an equation as, e.g., ‘ = [a U b] by adding ‘ = b _ a ^ X‘ to the transition relation fixes ‘ such that it behaves either as [a U b] or [a <u>U</u> b].

Moreover, the formulas of the second part of the above theorem show how we can assure that the newly generated variables ‘$_i$ can be fixed to be either the strong or the weak version of an operator by adding additional fairness constraints. These fairness constraints distinguish between the two solutions of the fixpoint equation and select safely one of both solutions.

```
function tableau( )
  case of
    is_prop(')      : return (fg; ');
    : '             : (E1; '1)  tableau(');
                      return (E1; : '1);
    ' ^             : (E1; '1)  tableau('); (E2;  1)  tableau( );
                      return (E1 [ E2; '1 ^  1);
    ' _             : (E1; '1)  tableau('); (E2;  1)  tableau( );
                      return (E1 [ E2; '1 _  1);
    X '             : (E1; '1)  tableau('); ‘ = new_var;
                      return (E1 [ f‘ = X '1g; ‘);
    [' B  ]         : (E1; '1)  tableau('); (E2;  1)  tableau( ); ‘ = new_var;
                      return (E1 [ E2 [ f‘ = ['1 B  1]g; ‘);
    [' U  ]         : (E1; '1)  tableau('); (E2;  1)  tableau( ); ‘ = new_var;
                      return (E1 [ E2 [ f‘ = ['1 U  1]g; ‘);

function trans(‘;  )
  case of
    ‘ = X '         : return ‘ = X ';
    ‘ = [' B  ]     : return ‘ = :  ^ (' _ X‘);
    ‘ = [' U  ]     : return ‘ =  _ ' ^ X‘;

function fair(‘;  )
  case of
    ‘ = X '         : return T;
    ‘ = [' B  ]     : return GF(‘ _ ' _  );
    ‘ = [' U  ]     : return GF(‘ ! : ' _  );

function Tableau( )
  (f‘1 = '1; : : : ; ‘n = 'ng;  I) := tableau(NNF( ));
   R := V_{i=1}^{n} trans(‘i; 'i);
   F := V_{i=1}^{n} fair(‘i; 'i);
  return A9 (f‘1; : : : ; ‘bg;  I;  R;  F);
```

Fig.2. Algorithm for translating LTL to ! -automata

It is clear that the equations in the second part of the theorem tell us how to replace the definitions $‘_i = '_i$ by an equivalent ! -automaton. For example, the definition $‘_1 =$ Ga is replaced with the formula:

$$[8t:‘_1^{(t)} = a^{(t)} \wedge ‘_1^{(t+1)}] \wedge 8t_1:9t_2:a^{(t_1+t_2)} \;\; ! \;\; ‘_1^{(t_1+t_2)}$$

As the introduced variables $‘_i$ occur under an existential quantifier, this formula corresponds directly to a generalized Büchi automaton. Hence, the following theorem holds:

Theorem 2 (Translating LTL to ! -Automata). *For the algorithm given in Figure 2, the following holds for any* LTL *formula and for* $(E;\Psi) =$ tableau( )*:*

- *jEj 2 $O(j\ \ j)$ and $j\Psi j$ + $\mathsf{P}_{‘_i = '_i 2 E}\, j\ '_i j$ 2 $O(j\ \ j)$, which implies linear runtime in terms of $j\ \ j$ of* tableau( )
- *Ψ is a propositional formula*
- *for each $‘_i\ =\ '_i\ 2\ E$, $'_i$ is a formula with exactly one temporal operator that occurs on top of $'_i$*
- *$'_i$ contains at most the variables that occur in plus the variables $‘_1,\ldots,‘_{i-1}$*
- $=\ 9‘_1 ::: ‘_n{:}\Psi\ \wedge\ \bigvee_{i2I}$ G$[‘_i\ =\ '_i]$

The result of the function Tableau *results in an equivalent generalized Büchi automaton with the initial states Ψ, transition relation* $\bigvee_{‘_i = '_i 2 E}$ trans$(‘_i;\ '_i)$ *and acceptance condition* $\bigvee_{‘_i = '_i 2 E}$ fair$(‘_i;\ '_i)$.

The construction yields in an automaton with $2^{O(j\ \ j)}$ states and an acceptance condition of length $O(j\ \ j)$. Note that the constructed $!$-automaton is in general nondeterministic. This can not be avoided, since deterministic Büchi automata are not as expressive as nondeterministic ones [30].

For our example FGa $!$ GFa, we derive the following $!$-automaton:

$$
\begin{array}{l}
9‘_1\ ‘_2\ ‘_3\ ‘_4{:}\ \mathrm{h}\ ‘_2^{(0)}\,"\ !\ ‘_4^{(0)}\ \mathrm{i}\ \wedge \\
8t{:}\ \mathrm{0\,h}\ \begin{array}{l} ‘_1^{(t)} = a^{(t)} \wedge ‘_1^{(t+1)} \wedge ‘_2^{(t)} = ‘_1^{(t)} \_\ ‘_2^{(t+1)} \wedge\ \# \\ ‘_3^{(t)} = a^{(t)} \_\ ‘_3^{(t+1)} \wedge ‘_4^{(t)} = ‘_3^{(t)} \wedge ‘_4^{(t+1)} \end{array} \wedge\ \mathrm{i} \quad 1 \\
\mathrm{@\,h}\ \mathrm{h}\ 8t_1{:}9t_2{:}a^{(t_1+t_2)}\ !\ ‘_1^{(t_1+t_2)}\ \mathrm{i} \wedge \mathrm{h}\ 8t_1{:}9t_2{:}‘_2^{(t_1+t_2)}\ !\ ‘_1^{(t_1+t_2)}\ \mathrm{i} \wedge\ \mathrm{A} \\
\quad 8t_1{:}9t_2{:}‘_3^{(t_1+t_2)}\ !\ a^{(t_1+t_2)} \wedge 8t_1{:}9t_2{:}‘_3^{(t_1+t_2)}\ !\ ‘_4^{(t_1+t_2)}
\end{array}
$$

As another example, consider the translation of a property that is to be verified for the single pulser circuit [40]:

$$\mathsf{G}[:i \wedge \mathsf{X}i\ \ !\ \ \mathsf{X}([o\ \mathsf{B}\ (:i \wedge \mathsf{X}i)]\ \_\ [o\ \mathsf{W}\ (:i \wedge \mathsf{X}i)])]$$

The formula specifies that after a rising edge of the input i, the output o must hold at least once before or at the time where the next rising edge of i occurs. The translation begins with the abbreviation of the subformulas starting with temporal operators. We obtain the definitions $‘_0$:= Xi, $‘_1$:= $[o$ B $(:i \wedge ‘_0)]$, $‘_2$:= $[o$ W $(:i \wedge ‘_0)]$, $‘_3$:= X$(‘_1\ _\ ‘_2)$, and $‘_4$:= G$[:i \wedge ‘_0\ \ !\ \ ‘_3]$. Replacing these definition with the preproven theorems, we finally end up with the following generalized Büchi automaton:

$$
\begin{array}{l}
9‘_0\ ‘_1\ ‘_2\ ‘_3\ ‘_4{:} \\
\quad ‘_4^{(0)} \wedge\ \mathrm{0\,h}\ \ \mathrm{i\ \ h}\ \ \mathrm{i}\quad 1 \\
8t{:}\ \mathrm{B\,h}\ ‘_0^{(t)} = i^{(t+1)} \wedge ‘_1^{(t)} = :(:i^{(t)} \wedge ‘_0^{(t)}) \wedge (o^{(t)} \_\ ‘_1^{(t+1)}) \wedge\ \mathrm{i\ h}\ \mathrm{C} \\
\quad \mathrm{B}\ ‘_2^{(t)} = \ :i^{(t)} \wedge ‘_0^{(t)}\ )\ o^{(t)}\ ‘_2^{(t+1)} \wedge ‘_3^{(t)} = ‘_1^{(t+1)} \_\ ‘_2^{(t+1)} \wedge\ \mathrm{C} \wedge \\
\quad \mathrm{@\,h}\ ‘_4^{(t)} = (:i^{(t)} \wedge ‘_0^{(t)}\ !\ ‘_3^{(t)}) \wedge ‘_4^{(t+1)}\ \mathrm{i} \wedge\ \mathrm{A} \\
\quad (8t_1{:}9t_2{:}‘_1^{(t_1+t_2)}\ \_\ o^{(t_1+t_2)}\ \_\ :i^{(t_1+t_2)} \wedge ‘_0^{(t_1+t_2)}) \wedge\ ! \\
\quad (8t_1{:}9t_2{:}‘_2^{(t_1+t_2)}\ \_\ :i^{(t_1+t_2)} \wedge ‘_0^{(t_1+t_2)})
\end{array}
$$

We have implemented a HOL conversion that computes a corresponding generalized Büchi automaton as explained above and then proves the equivalence with our preproven theorems of Theorem 1. It is easily seen that the computation of the generalized Büchi automaton is done in linear runtime wrt. the length of the given formula.

6 Applications

The Prosper project focuses on reducing the gap between formal verification and industrial aims and needs. Although formal methods have proven their usefulness and importance in academia for years, they are only rarely applied in industrial environments. One reason for this is that existing proof tools require profound knowledge in logic. Since only a very few system engineers have this expertise, formal methods are often not applied at all. Another reason is the lack of automation. Many proof tools need considerable user interaction which complicates its usage and slows down the complete design cycle.

Within Prosper, examples of proof tools are being produced which provide user friendly access to formal methods. An open proof architecture allows the integration of different verification tools in a uniform higher order logic environment. Besides providing easy and consistent access to these tools, a high degree of automation can be achieved by integrating various decision procedures as plug-ins. Examples of already integrated decision procedures are a Boolean tautology checker based on BDDs, PROVER from Prover Technology, and the CTL model checker SMV.

As mentioned before, the translation of LTL formulas to ! -automata as described in the previous section can be used to model check LTL formulas with a CTL model checker. Using HOL's CTL model checker plug-in which has been developed as part of the Prosper project, our transformation procedure shows how LTL model checking can be performed directly within HOL.

6.1 The Prosper Plug-In Interface

The basis for a uniform integration of different proof tools in HOL is Prosper's plug-in Interface [41, 42]. It provides an easy to use and formally specified communication mechanism between HOL and its various proof backends. Communication is either based on Internet sockets or on local Unix pipes, depending on the machine where the proof backend is running. The plug-in interface can roughly be divided into two parts. One part that manages communication with HOL (proof engine side) and another part which is responsible for the plugged in proof backend (plug-in side). If Internet sockets are used for communication, both sides of the plug-in might run on different machines. The proof engine side is directly linked to HOL whereas the plug-in side runs on the same machine as the proof backends. If more than one plug-in is running, each back end is linked by a separate plug-in interface. This allows the integration of arbitrary tools running either locally on the same machine or widespread over the world connected via Internet sockets. Overall, this leads to the sketch in Fig. 3

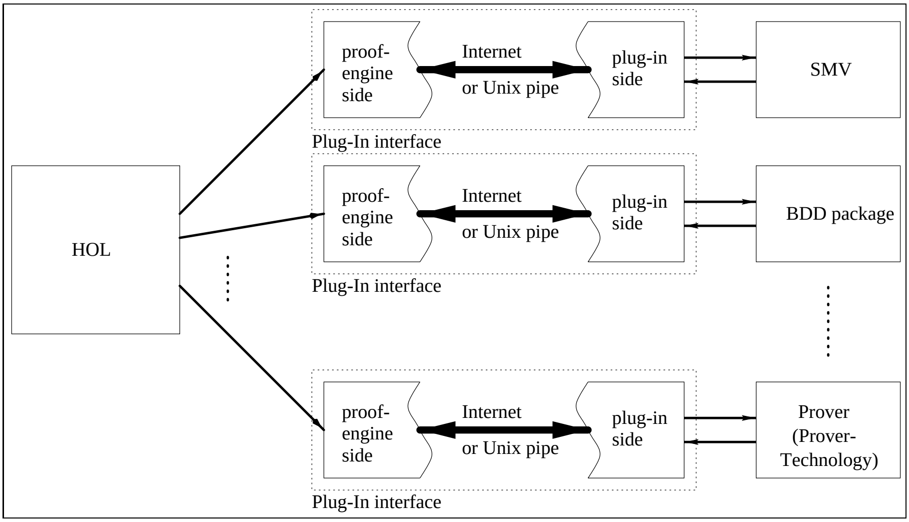


Fig.3. Sketch of the Prosper plug-in Interface

6.2 The CTL Model Checker Plug-In

As part of the Prosper project, we have integrated the SMV model checker [2] into HOL via Prosper's plug-in interface. SMV has been chosen since it is one of the widest spread and most used model checkers. Moreover, SMV is freely available and can therefore be distributed with HOL.

To be able to reason about SMV programs, the SMV language defined in [2] has been deeply embedded in HOL[3]. According to the grammar definition given in [2], a new HOL type is defined for each non-terminal, i.e., the following HOL types are defined:

```
smv_constant      smv_id            smv_expr
smv_ctl           smv_type_enum     smv_type
smv_var_decl      smv_alhs          smv_assign_decl
smv_define_decl   smv_declaration   smv_moduleports
smv
```

Having the SMV grammar in mind, these data types are defined in a straightforward manner, e.g., `smv_constant` is defined as

```
val smv_constant = define_type
    {name = "smv_constant",
     type_spec = `smv_constant = ATOM_CONSTANT of string
                               | NUMBER_CONSTANT of num
                               | FALSE_CONSTANT
                               | TRUE_CONSTANT`,
     fixities = [Prefix,Prefix,Prefix,Prefix]};
```

[3] http://goethe.ira.uka.de/~hoff/smv.sml

All other data types are defined in a similar way. At the moment, the language is only embedded syntactically. The semantics which is also given in [2] will be formalized in HOL soon.

A parser exists which converts SMV programs to their corresponding representation in HOL. Once the datatype has been created, it can be manipulated within HOL, or SMV can be invoked with the `MC` command. Calling SMV via the plug-in interface, the HOL datatype is converted into SMV readable format and passed to the model checker. If the specification is true, the HOL term T is returned to HOL, otherwise F is returned. Once the semantics of SMV programs has been formalized (e.g., by defining a predicate valid(*S*) which is true if and only if *S* satisfies its specification), the model checker plug-in can be adapted easily to return a theorem *j*= valid(*S*) instead of the HOL term T.

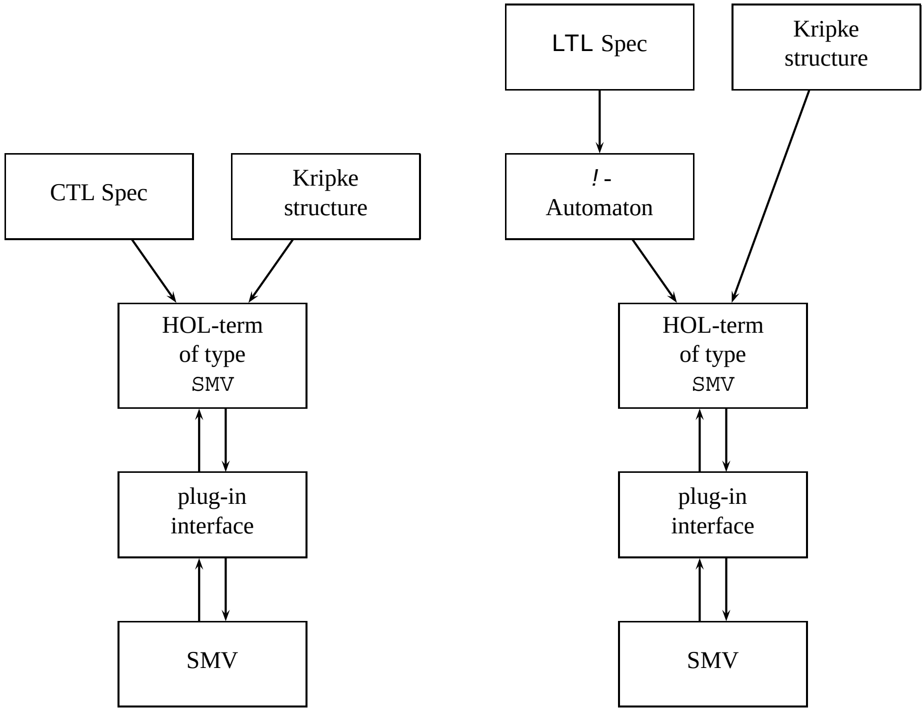


Fig.4. Invoking the SMV model checker via the Prosper plug-in interface. The left picture shows the usual call to SMV with a CTL specification. The right picture demonstrates how our transformation of LTL formulas to ! -automata can be used to model check LTL formulas with the same plug-in.

Internally, the SMV plug-in interface interacts with SMV by using standard I/O communication. This design decision has been made because it avoids changes in the SMV source code. Treating SMV as a black box tool, it can be easily upgraded when new versions of SMV are released.

Fig. 4 (left) shows how SMV is invoked with a CTL specification. Fig. 4 (right) shows how our procedure for transforming LTL formulas to !-automata can be used to model check LTL formulas with the same plug-in.

7 Conclusions and Future Work

We have described a translation procedure for converting LTL formulas to equivalent !-automata and its implementation in the HOL theorem prover. Together with the SMV plug-in this allows the usage of SMV as a decision procedure that can be conveniently called as a HOL tactic. As a result, temporal logic formulas given in LTL can now be used to specify and to verify the concurrent behavior conveniently by the model checker SMV, although this model checker is not directly able to handle LTL. The translation of LTL into !-automata by our HOL conversion runs in linear time and is also in practice very efficient since it is mainly based on preproven theorems of the LTL theory that we also provided.

References

[1] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[2] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[3] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[4] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Rajeev Alur and Thomas A. Henzinger, editors, *Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[5] A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S.C. Krishnan, R.K. Ranjan, T.R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. HSIS: A BDD-Based Environment for Formal Verification. In *ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, June 1994. San Diego Convention Center.

[6] R. K. Brayton, A. L. Sangiovanni-Vincentelli, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. K. Ranjan, T. R. Shiple, G. Swamy, T. Villa, G. D. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary. VIS: A system for verification synthesis. In *Computer-Aided Verification*, New Brunswick, NJ, July-August 1996.

[7] M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, C-35(12):1034–1044, December 1986.

[8] D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, 133 Part E(5):276–282, September 1986.

[9] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 5–20, Ottawa, Canada, April 1993. IFIP WG10.2, CHDL'93, IEEE COMPSOC, Elsevier Science Publishers B.V., Amsterdam, Netherland.

[10] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[11] N. Shankar. PVS: Combining specification, proof checking, and model checking. In M. Srivas and A. Camilleri, editors, *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, USA, November 1996. Springer Verlag.

[12] E.A. Emerson and J.Y. Halpern. ”sometimes” and ”not never” revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[13] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[14] K. Schneider. CTL and equivalent sublanguages of CTL*. In C. Delgado Kloos, editor, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 40–59, Toledo,Spain, April 1997. IFIP, Chapman and Hall.

[15] K. Schneider, T. Kropf, and R. Kumar. Why Hardware Verification Needs more than Model Checking. In *Higher Order Logic Theorem Proving and its Applications*, Malta, 1994.

[16] P. Loewenstein. Formal verification of state-machines using higher-order logic. In *IEEE/ACM International Conference on Computer Design (ICCD)*, pages 204–207, 1989.

[17] P. Loewenstein. A formal theory of simulations between infinite automata. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 227–246, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.

[18] K. Schneider, R. Kumar, and T. Kropf. Alternative Proof Procedures for Finite-State Machines in Higher-Order Logic. In J.J. Joyce and C.-J.H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 213–227, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[19] D. Eisenbiegler and R. Kumar. An Automata Theory Dedicated Towards Formal Circuit Synthesis. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 154–169, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

[20] K. Schneider and T. Kropf. A unified approach for combining different formalisms for hardware verification. In M. Srivas and A. Camilleri, editors, *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 202–217, Palo Alto, USA, November 1996. Springer Verlag.

[21] S. Agerholm and H. Schjodt. Automating a model checker for recursive modal assertions in HOL. Technical Report DAIMI IR-92, DAIMI, January 1990.

[22] J. von Wright. Mechanizing the temporal logic of actions in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Higher Order Logic Theorem Proving and its Applications*, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.

[23] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.

[24] F. Andersen and K.D. Petersen. Recursive Boolean Functions in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Higher Order Logic Theorem Proving and its Applications*, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.

[25] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, May 1989.

[26] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Horsholm, Denmark, March 1992.

[27] F. Andersen, K.D. Petersen, and J.S. Petterson. Program Verification using HOL-UNITY. In J.J. Joyce and C.-J.H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 1–16, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[28] K. Schneider. Translating linear temporal logic to deterministic ! -automata. In M.Pfaff and R. Hagelauer, editors, *GI/ITG/GMM Workshop Methoden des Entwurfs und der Verifikation digitaler Systeme*, pages 149–158, 1997.

[29] K. Schneider. Yet another look at LTL model checking. In *IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science, Bad Herrenalb, Germany, September 1999. Springer Verlag.

[30] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191, Amsterdam, 1990. Elsevier Science Publishers.

[31] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–107, New York, January 1985. ACM.

[32] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Conference on Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, New York, 1985. Springer-Verlag.

[33] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[34] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, pages 75–123, Altrincham, UK, 1987. Springer-Verlag.

[35] G.G de Jong. An automata theoretic approach to temporal logic. In K.G. Larsen and A. Skou, editors, *Workshop on Computer Aided Verification (CAV)*, volume 575 of *Lecture Notes in Computer Science*, pages 477–487, Aalborg, July 1991. Springer-Verlag.

[36] S. Safra. On the complexity of ! automata. In *IEEE Symp.on Foundations of Computer Science*, pages 319–327, 1988.

[37] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff'94*, 1994.

[38] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Conference on Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Standford, California, USA, June 1994. Springer-Verlag.

[39] K. Schneider. Model checking on product structures. In G.C. Gopalakrishnan and P.J. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 483–500, Palo Alto, CA, November 1998. Springer Verlag.

[40] S.D. Johnson, P.S. Miner, and A. Camilleri. Studies of the single pulser in various reasoning systems. In T. Kropf and R. Kumar, editors, *International Conference on Theorem Provers in Circuit Design (TPCD)*, volume 901 of *Lecture Notes in Computer Science*, pages 126–145, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.

[41] M. Norrish, L. Dennis, and R. Boulton. Prosper plug-in interface design. Prosper project report D3.2a, October 1998.

[42] M. Norrish, G. Collins, L. Dennis, and R. Boulton. Prosper plug-in interface user documentation. Prosper Project Report D3.2b, November 1998.

From I/O Automata to Timed I/O Automata

A Solution to the 'Generalized Railroad Crossing' in Isabelle/HOLCF

Bernd Grobauer[1?] and Olaf Müller[2]

[1] BRICS, Department of Computer Science, University of Aarhus, Denmark
[2] Institut für Informatik, TU München, 80290 München, Germany

Abstract. The model of timed I/O automata represents an extension of the model of I/O automata with the aim of reasoning about real-time systems. A number of case studies using timed I/O automata has been carried out, among them a treatment of the so-called *Generalized Railroad Crossing* (GRC). An already existing formalization of the meta-theory of I/O automata within Isabelle/HOLCF allows for fully formal tool-supported verification using I/O automata. We present a modification of this formalization which accomodates for reasoning about timed I/O automata. The guiding principle in choosing the parts of the meta-theory of timed I/O automata to formalize has been to provide all the theory necessary for formalizing the solution to the GRC. This leads to a formalization of the GRC, in which not only the correctness proof itself has been formalized, but also the underlying meta-theory of timed I/O automata, on which the correctness proof is based.

1 Introduction

The model of timed I/O automata (see e.g. [10, 8]) represents an extension of the model of I/O automata (cf. [11]) with the aim of reasoning about real-time systems. A number of case studies using timed I/O automata has been carried out, among them a treatment of the so-called *Generalized Railroad Crossing* (GRC) [9]. As experience shows, a formal method is only practical with proper tool support. In contrast to many comparable formalisms, where the specification language is tuned to make certain verification tasks decidable, a timed I/O automaton is in general an infinite transition systems. Hence the only comprehensive and generic tool support for proofs within the model of timed I/O automata is (interactive) theorem proving; of course additional support using e.g. model checking for deciding manageable subproblems could (and even should) be provided.

The aim of our work is to create an environment for reasoning about timed I/O automata and to apply it to a formalization of the GRC. The starting point is an existing framework for reasoning about untimed I/O automata [12, 13, 14] in

[?] BRICS: Basic Research in Computer Science, Centre of the Danish National Research Foundation.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 273{289, 1999.

the theorem prover Isabelle/HOLCF [15]. It provides an extensive formalization of the theory of I/O automata, e.g. all the proof principles which are offered for actual applications have been shown to be sound. Such a formalization of a formal method's meta-theory not only rules out potential sources of unsoundness, it also allows one to *derive* new proof principles instead of hardwiring them. Further it leads to a deeper understanding of the theory itself; changes to the theory which harmonize with the chosen formalization can easily be carried out and their effects evaluated. In a sense the presented work is 'living proof' for the last claim: one of its main strengths is the simplicity of extending the existing framework from untimed to timed systems.

The guiding principle in choosing the parts of the theory of timed I/O automata to formalize has been to provide all the theory necessary for formalizing the GRC as presented in [9]. In contrast to our GRC solution, Archer and Heitmeyer, who carried out an alternative formalization [2] in PVS [18], do not treat the meta-theory at all and so far only deal with invariants without taking the step to simulations. In the following we are going to describe both the formalization of the theory of timed I/O automata in Isabelle/HOLCF and the subsequent formalization of the GRC.

After some preliminaries about Isabelle/HOLCF which are treated in Section 2, Section 3 gives an introduction to the theory of timed I/O automata. In Section 4 the formalization of the theory of timed I/O automata in Isabelle/HOLCF is described. Its application to the formalization of the GRC is sketched in Section 5.

2 Isabelle/HOLCF

Isabelle [19] is a generic interactive theorem prover; we use only its instantiation with HOLCF [15], a conservative extension of higher order logic (HOL) with an LCF-like domain theory. Following a methodology developed in [13], the simpler HOL is used wherever possible, only switching to LCF when needed.

Several features of Isabelle/HOL help the user to write succinct and readable theories. There is a mechanism for ML-style definitions of inductive data types [4]. For every definition of a recursive data type, Isabelle will automatically construct a type together with the necessary proofs that this type has the expected properties. Another feature working along the same lines allows the inductive definition of relations. Built on top of the data type package is a package which provides for extensible record types [16]. The keywords introducing the cited language features are **datatype**, **inductive** and **record** respectively.

Isabelle's syntax mechanisms allow for intuitive notations; for example set comprehension is written as $\{e \mid P\}$, a record s which contains a field foo can be updated by writing $s(\!|\mathsf{foo} := bar|\!)$. A similar notation is used for function update: $f(e_1 := e_2)$ denotes a function which maps e_1 onto e_2 and behaves like f on every other value of its domain.

The type constructor for functions in HOL is denoted by $\Rightarrow$, the projections for pairs by fst and snd. Simple non-recursive definitions are marked with the keyword **defs**, type definitions with **types**, and theorems proven in Isabelle with the keyword **thm**.

3 The Theory of Timed I/O Automata

A timed I/O automaton A is a labelled transition system, where the labels are called *actions*. These are divided into *external* and *internal* actions, denoted by $ext(A)$ and $int(A)$, respectively. The external actions are divided into *input* and *output* actions (denoted by $inp(A)$ and $out(A)$) and a collection of special *time-passage* actions $\{\nu(t) \mid t \in T\}$. Here T is a dense time domain which usually is taken to be $\mathbb{R}^{>0}$. Hence a timed I/O automaton A can be specified by defining

- a set $states(A)$ of states
- a nonempty set $start(A) \subseteq states(A)$
- an action signature $sig(A) = (inp(A), out(A), int(A))$ where $inp(A)$, $out(A)$ and $int(A)$ are pairwise disjoint. None of the actions in the signature is allowed to be of the form $\nu(t)$ with $t \in T$. We call $vis(A) := inp(A) \cup out(A)$ the set of *visible* actions; $ext(A)$ thus can be written as $vis(A) \cup \{\nu(t) \mid t \in T\}$
- a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$, where $acts(A)$ is defined as $ext(A) \cup int(A)$. Let $r \xrightarrow{a}_A s$ denote $(r, a, s) \in trans(A)$

There are a number of additional requirements a timed I/O automaton must satisfy, which give some insight about the intuition behind the definitions: For example a time-passage action $\nu(t)$ is supposed to stand for the passage of time t. Obviously if t time units pass and then another t' units, this should have the same effect as if $t+t'$ time units passed in one step. Therefore it is required that if $s \xrightarrow{\nu(t)}_A s'$ and $s' \xrightarrow{\nu(t')}_A s''$ then $s \xrightarrow{\nu(t+t')}_A s''$.

Timed I/O automata can be combined by *parallel composition*, which is denoted by $\|$. A step of a composed automaton $A\|B$ is caused either by an internal step of A or B alone, a combined step with an input action common to A and B, or a communication between A and B via an action that is input action to A and output action to B (or vice versa).

Executions and Traces

The notion of behavior used for timed I/O automata is *executions* and, as behavior observable from the outside, *traces*.

As for executions, two models are commonly used. The first of them views an execution ex as the continuous passage of time with timeless actions occurring at discrete points in time:

$$ex = \omega_0 a_1 \omega_1 a_2 \omega_2 \ldots \quad .$$

Here the ω_i are so-called *trajectories*, that is mappings from an interval of the time domain into the state-space of the automaton — the exact definition of a trajectory makes sure that it defines only time-passage that is compatible with the specification of the automaton. These executions are called *timed* executions. We however are going to formalize a model which is called *sampled* executions in [10]. Here an execution is a sequence in which states and actions alternate:

$$ex = s_0 a_1 s_1 a_2 s_2 :::$$

where an action a_i is either a discrete action or a time-passage action. Sampling can be seen as an abstraction of timed executions: instead of trajectories, which hold information about the state of an automaton for every point of time, explicit time-passage steps occur in executions.

The total amount of time which has passed up to a certain point in an execution can be deduced by adding up all the time-values which parameterized the time-passage actions that occured so far. Notice that the execution model includes executions which intuitively speaking do not make sense | since time cannot be stopped, only executions with an infinite total amount of time-passage do. Executions with an infinite total amount of time-passage are called *admissible*.

An execution gives rise to a trace in a straightforward way by first filtering out the invisible actions, suppressing the states and associating each visible action with a time stamp of the point of time of its occurrence. Then the resulting sequences of (visible) actions paired with a time stamp is paired with the least upper bound of the maximal time value reached in the execution (a special symbol 1 is used in case of an admissible execution). Traces that stem from admissible executions are themselves called admissible.

The observable behavior of a timed I/O automaton A is viewed to be the set of its traces $traces(A)$ | this may sometimes be restricted to certain kinds of traces, e.g. admissible traces. If the signatures of two automata A and C agree on the visible actions, then C is said to implement A iff $traces(C)$ $traces(A)$.

Proof Principles

The most important proof methods for timed I/O automata are invariance proofs and simulation proofs. The former is used to show that a predicate over $states(A)$ holds for every reachable state in an automaton A. Simulation proofs are used to show that an automaton C implements an automaton A; they are based on a theorem about timed I/O automata which says that if there exists a so-called simulation relation S $states(C)$ $states(A)$, then C implements A.

4 Formalizing Timed I/O Automata in HOLCF

The treatment of the GRC as presented in [9] is based upon the model of timed I/O automata. Apart from the proof principles mentioned above, namely invariance proofs and simulation proofs, a further proof principle is used to show a certain kind of execution inclusion via a simulation relation. Compositional reasoning is not employed. In the following we give an overview of the formalization of the underlying theory of timed I/O automata in Isabelle. Often only minor changes to the already existing formalization of the theory of I/O automata had to be made, we will mostly concentrate on these changes and additions | see [1] for the complete formalization. Further information about the original formalization of I/O automata can be found in [13]. All in all about one and a half months were spent in formalizing the necessary meta-theory | two weeks for changing the underlying theory of I/O automata and four weeks for the additions described in Section 4.4; some of this time however was spent by the first author reacquainting himself with Isabelle.

4.1 Timed I/O Automata

All the proofs that have been carried out in order to formalize the necessary fragment of the theory of timed I/O automata only require the time domain T to be an ordered group. Therefore we keep the whole Isabelle theory of timed I/O automata parametric with respect to a type representing an ordered group. This can be achieved in an elegant way using the mechanism of *axiomatic type classes* [22] of Isabelle: a type class timeD is defined which represents all types that satisfy the axioms of an ordered group with respect to a signature $h\ ;\ ;\ ;0;\ i$. The notation $_{\text{timeD}}$ signifies that is such a type.

Actions of timed I/O automata can either be discrete actions or time-passage actions; we introduce a special data type:

datatype (; $_{\text{timeD}}$)action = Discrete j Time

In the following a discrete action Discrete *a* will be written as ◁*a*▷, a time-passage action Time *t* as (*t*).

We further chose to make timing information explicit in the states, as for example done in [10]. There only one special time-passage action is used, which suffices as time-passage can be read from the states. We, however, keep timing information both in the states and in the actions | we argue that specifications of timed I/O automata will gain clarity. Therefore states will always be of type (; $_{\text{timeD}}$)state, where

record (;)state =
content ::
now :: $_{\text{timeD}}$

With these modifications in mind, the definitions made in Section 3 give rise to the following type definitions in a straightforward way:

types signature = (set set set)
(; ; $_{\text{timeD}}$)transition = (;)state (;)action (;)state
(; ; $_{\text{timeD}}$) tioa = signature (;)state set (; ;)transition set

Thus, (; ;)tioa stands for a timed I/O automaton where discrete actions are of type , the state contents (i.e. the state without its time stamp) of type and the used time domain of type , which is required to be in the axiomatic type class timeD. For tioa and signature also selectors have been defined, namely

{ inputs, outputs and internals, giving access to the different kinds of (discrete) actions that form a signature, together with derived functions like visibles *s* = inputs *s* [outputs *s*, which are the visible discrete actions,

{ sig-of, starts-of and trans-of giving access to the signature, the start states and the transition relation of a timed I/O automaton.

The additional requirements on timed I/O automata as introduced in Section 3 are formalized as predicates over the transition relation of an automaton. Since we have chosen to carry time stamps within the states, an additional well-formedness condition has to be formulated:

$$
\begin{array}{ll}
\textsf{defs} & \textsf{well-formed-trans}\ A\ =\\
 & 8(s;a;r)\ 2\ (\textsf{trans-of}\ A) : ::: \wedge\ (\textsf{case}\ a\ \textsf{of}\\
 & \qquad\qquad\qquad\qquad \lhd a^0 \rhd\ \ !\ \ \textsf{now}\ s = \textsf{now}\ r\\
 & \qquad\qquad\qquad\quad j\ \ (t)\ \ !\ \ \textsf{now}\ r = (\textsf{now}\ s)\quad t)
\end{array}
$$

This and the other requirements on timed I/O automata are combined in a predicate TIOA over type tioa.

4.2 Executions and Traces

As mentioned above we are going to formalize sampled executions. In [10] it has been shown that reachability is the same for both execution models. Further for each sampling execution there is a timed execution that gives rise to the same trace and vice versa. Since the sampling model preserves traces and reachability, it is sufficient for our purposes. After all we are not so much interested in meta-theory dealing with the completeness of refinement notions (cf. [10]), but want to provide a foundation for actual verification work. Executions thus boil down to sequences in which states and actions alternate. Only technical modifications to the representation of executions used in the already existing formalization of I/O automata had to be made. Here the LCF-part of HOLCF comes in | lazy lists are used to model possibly infinite sequences (cf. [7]). Consider the following type declarations:

$$
\begin{array}{rcl}
\textsf{types}\ (\ ;\ ;\ _{\textsf{timeD}})\textsf{pairs} & = & ((\ ;\ )\textsf{action}\quad (\ ;\ )\textsf{state})\textsf{Seq}\\
(\ ;\ ;\ )\textsf{execution} & = & (\ ;\ )\textsf{state}\quad (\ ;\ ;\ )\textsf{pairs}\\
(\ ;\ )\textsf{trace} & = & ((\ ;\ )\textsf{action}\qquad)\textsf{Seq}
\end{array}
$$

Here Seq is a type of lazy lists specialized for a smooth interaction between HOL and HOLCF (cf. [13]). An execution consists of a start state paired with a sequence of action/state pairs, whereas a trace is a sequence of actions paired with a time stamp (we chose not to include the end time reached in the execution which gave rise to the trace, since no information useful for actual verification work is added by it). All further definitions in the theory of timed I/O automata are formalized with functions and predicates over the types defined above. For example is-exec-frag *A ex* checks whether *ex* is an execution fragment of an automaton *A*. Its definition is based upon a continuous LCF-function is-exec-frag$_c$.

To explain our formalization of an execution fragment, it suffices to display some properties of is-exec-frag which have been derived automatically from its definition:

$$\textbf{thm}\ \ \text{is-exec-frag}\ A\ (s,[])$$
$$\textbf{thm}\ \ \text{is-exec-frag}\ A\ (s,(a,r)\text{\^{}}ex) \ = \ ((s,a,r) \in \text{trans-of}\ A \wedge \text{is-exec-frag}\ A\ (r,ex))$$

Here [] stands for the empty sequence and ^ for the cons-operation.

Further definitions include a function mk-trace which maps an execution to the trace resulting from it, executions and traces which give the set of executions resp. traces of an automaton.

Reachability can be defined via an inductive definition:

$$\textbf{inductive}\quad \frac{s \in \text{starts-of}\ C}{s \in \text{reachable}\ C} \qquad \frac{s \in \text{reachable}\ C \quad (s,a,t) \in \text{trans-of}\ C}{t \in \text{reachable} C}$$

4.3 Invariant Proofs and Simulation Proofs for Trace Inclusion

Neither showing the correctness of the proof principle for invariants nor that of simulation proofs required significant changes of the formalization for untimed I/O automata. An invariant is defined as

$$\textbf{defs}\qquad \text{invariant}\ A\ P = \forall s.\ \text{reachable}\ A\ s \ \Longrightarrow\ P(s)$$

The principle of invariance proofs, namely

$$\textbf{thm}\quad \frac{\begin{array}{l}\forall s \in \text{starts-of}\ A.\ P(s)\\ \forall s\ a\ t.\ \text{reachable}\ A\ s \ \wedge\ P(s)\ \wedge\ s \xrightarrow{a}_A t \ \Longrightarrow\ P(t)\end{array}}{\text{invariant}\ A\ P}$$

is easily shown by rule induction over reachable. The correctness of simulation proofs has been shown for three frequently used kinds of simulation relations, namely *refinement mappings*, *forward simulations* and *weak forward simulations*. Since it turned out that *weak forward simulations* are preferable in practice (see section 5.4), we will restrict our presentation to this proof principle. All the simulation relations mentioned are based on the notion of a *move*: Let $s \overset{\langle a,t\rangle}{\underset{ex}{\Longrightarrow}}_A r$ denote an a-move from state s to state r in an automaton A by way of an execution fragment ex. The definition of a move requires, that the execution fragment ex has s as first and r as last state; further the trace arising from ex is either the sequence holding only $\langle a,t\rangle$ if a is a visible action, or the empty sequence if a is not visible. The intuition is that A performs an action a and possibly some invisible actions before and after that in going from s to r.

A forward simulation R between an automaton C and an automaton A basically says: any step $s \xrightarrow{a}_C r$ from a reachable state s of C can be matched by a corresponding move of A:

$$
\begin{array}{ll}
\textsf{defs} & \textsf{is-w-simulation}\ R\ C\ A = \\
& \quad (8s\ u{:}\ u\ 2\ R[s] =)\ \textsf{now}\ u = \textsf{now}\ s)\ \hat{} \\
& \quad (8s\ 2\ \textsf{starts-of}\ C{:}\ R[s]\ \backslash\ \textsf{starts-of}A\ 6\!=\ ;)\ \hat{} \\
& \quad (8s\ s^0\ r\ a{:}\ \textsf{reachable}\ C\ s \\
& \qquad\quad \hat{}\ s\ \overset{a}{-}!_{C}\ r \\
& \qquad\quad \hat{}\ (s; s^0)\ 2\ R \\
& \qquad\quad \hat{}\ \textsf{reachable}\ A\ s^0 \\
& \qquad\quad =)\ 9r^0\ ex{:}\ (r; r^0)\ 2\ R\ \ \hat{}\ \ s^0{}_{ex} \overset{ha;now\ r'i}{=)}{}_{A}\ r^0)
\end{array}
$$

The first line of the given definition requires a weak forward simulation R to be *synchronous*, i.e. to relate only states with the same time stamp ($R[s]$ denotes the image of s under R). The second line requires the set of all states related to some start state s of C to contain at least one start state of A. The requirement that s^0 be reachable in A and s reachable in C characterizes a *weak* forward simulation.

The corresponding proof principle, which has been derived in Isabelle, is

$$
\textsf{thm}\qquad \frac{\textsf{visibles}\ C = \textsf{visibles}\ A \quad \textsf{is-w-simulation}\ R\ C\ A}{\textsf{traces}\ C \quad \textsf{traces}\ A}
$$

At first it is surprising that showing the correctness of simulation relations required hardly any changes to the corresponding proof for untimed I/O automata. In particular no information about the time domain is needed. The point is that *synchronous* simulation relations restrain time-passage so much that time-passage actions do not differ significantly from other invisible actions in this context.

4.4 Simulation Proofs for Execution Inclusion

In the GRC case study carried out with timed I/O automata in [9], inclusion of executions under projection to a common component automaton is derived from the existence of a simulation relation between two automata. In doing so, the authors refer to general results about composition of timed automata. When trying to formalize this step we realized that one of the automata involved cannot be regarded as constructed by parallel composition. Therefore results about the composition of timed automata cannot be directly applied; further the very notion of projecting executions to a component automaton is not well-defined.

In order to formalize the meta-theory used for solving the GRC we tried to find a suitable reformulation of the employed concept, which is as well interesting in its own right. The basic idea is to find a notion of projecting executions to parts of an automaton which is compatible with the concept of simulation proofs: inclusion of executions under projection to a common part of two automata should be provable by exhibiting a simulation relation.

In which setting will execution inclusion with respect to some projection be employed? Say timed I/O automata are to be used for specifying a controller which influences an environment. Both the environment and the implementation of the controller will be modelled using automata | their parallel composition

C describes the actual behavior of the controlled environment. Trying to show correctness via the concept of simulation proofs, the *desired* behavior of the controlled environment will be specified as another automaton A. For constructing A, the automaton specifying the environment is likely to be reused, i.e. A arises from modifying this automaton such that its behavior is restricted to the desired behavior. Hence the state space of A will contain the environment's state space, which can be extracted by a suitable projection.

The notion of observable behavior one wants to use in this case might very well be executions rather than traces: executions hold information also about states, and it often is more natural to define desired behavior by looking at the states rather than visible actions only. What one wants to do is to define projections for both automata, which extract the environment. Using these projections, executions of both automata can be made comparable. As the following theorem shows, a simulation indeed implies inclusion of executions under the given projections, if these fulfill certain requirements:

$$\textsf{thm}\quad \frac{\begin{array}{l}\textsf{is-w-simulation}\ R\ C\ A\\ \forall u\ u'.\ (u,u') \in R \Longrightarrow p_c u = p_a u'\\ \forall a\ s\ s'.\ a \notin actS \wedge s \xrightarrow{\langle a\rangle}_A s' \Longrightarrow p_a s = p_a s'\\ actS \subseteq \textsf{visibles}(\textsf{sig-of}\ A) \cap \textsf{visibles}(\textsf{sig-of}\ C)\end{array}}{\begin{array}{l} ex \in \textsf{executions}\ C \Longrightarrow\\ \qquad \exists ex'.\ \textsf{exec-proj}\ A\ p_a\ actS\ ex' = \textsf{exec-proj}\ C\ p_c\ actS\ ex\end{array}}$$

Here p_c and p_a are projections from the state space of C and A on a common subcomponent. The set $actS$ is a subset of the shared visible discrete actions of C and A (see the fourth premise). The third premise of the theorem expresses that $actS$ has to include all the actions which in A (describing the desired behavior) affect the part of the state space that is projected out. The second premise gives a wellformedness condition of the simulation relation R with respect to p_c and p_a — only states which are equal under projection may be related.

It remains to clarify the rôle of exec-proj (actually defined by an elaborate continuous function $\textsf{exec-proj}_c$; all the following properties of exec-proj have been proven correct by reasoning over $\textsf{exec-proj}_c$ within the LCF-part of HOLCF). The function exec-proj in effect defines a new notion of observable behavior based on executions. Certainly one would expect exec-proj $A\ p\ actS$ to use projection p on the states of a given execution and to remove actions not in $actS$:

$$\begin{array}{lll}\textsf{thm} & a \in actS \Longrightarrow & \textsf{exec-proj}\ A\ p\ actS\ (s, (\langle a\rangle, s')\hat{}xs)\\ & & = (p\ s, (\langle a\rangle, p\ s')\hat{}(\textsf{snd}\ (\textsf{exec-proj}\ A\ p\ actS\ (s', xs))))\\ \textsf{thm} & a \notin actS \Longrightarrow & \textsf{exec-proj}\ A\ p\ actS\ (s, (\langle a\rangle, s')\hat{}xs)\\ & & = (p\ s, \textsf{snd}\ (\textsf{exec-proj}\ A\ p\ actS\ (s', xs)))\end{array}$$

However it is also necessary to abstract over subsequent time-passage steps:

$$\begin{array}{ll}\textsf{thm} & \textsf{exec-proj}\ A\ p\ actS\ (s, (\nu(t'), s')\hat{}(\nu(t''), s'')\hat{}xs)\\ & \quad = \textsf{exec-proj}\ A\ p\ actS\ (s, (\nu(t' + t''), s'')\hat{}xs)\end{array}$$

By removing actions that are not in $actS$ it can happen that time-passage steps which before were separated by discrete actions appear in juxtaposition. Hence we further require that

$$\begin{aligned}
\textbf{thm}\quad & a \in actS \Longrightarrow \text{exec-proj}\ A\ p\ actS\ (s, (\ (t'), s')\hat{}(\lhd a \rhd, s'')\hat{}xs) \\
& \quad = (p\ s, (\ (t'), p\ s')\hat{}(\lhd a \rhd, p\ s'')\hat{}(\text{snd}\ (\text{exec-proj}\ A\ p\ actS\ (s, xs)))) \\
\textbf{thm}\quad & a \notin actS \Longrightarrow \text{exec-proj}\ A\ p\ actS\ (s, (\ (t'), s')\hat{}(\lhd a \rhd, s'')\hat{}xs) \\
& \quad = \text{exec-proj}\ A\ p\ actS\ (s, (\ (t'), s')\hat{}xs)
\end{aligned}$$

Notice that the notion of observable behavior implied by exec-proj may not be suitable for all applications. It suffices for the GRC, because in all the automata specified there time-passage leaves everything of the state unchanged except from the time stamp. Only because of this we can abstract away from subsequent time-passage steps: all the states dropped by this abstraction differ only in their time stamp, i.e. no information is lost.

The proof of the correctness theorem displayed above is rather involved. However for meta-theoretic proofs, which only have to be carried out once and for all, complicated proofs are acceptable. The interaction of HOL and LCF in the formalization of (timed) I/O automata is such that users can carry out actual specifications and their verification with (timed) I/O automata strictly in the simpler logic HOL (cf. the deeper discussion in [12]). For the proof we also required assumptions about timeD — as mentioned above it was sufficient to know that every type in class timeD is an ordered group with respect to $\langle\ ,\ ,\ , 0,\ \rangle$.

5 A Formalization of the ‘Generalized Railroad Crossing’

In the specification of a solution to the GRC problem using timed I/O automata we closely follow the presentation of [9], formalizing invariant proofs and a simulation proof that were presented only in an informal way. In the following we give an overview over the GRC and its formalization within our framework. The complete Isabelle theories can be found in [1].

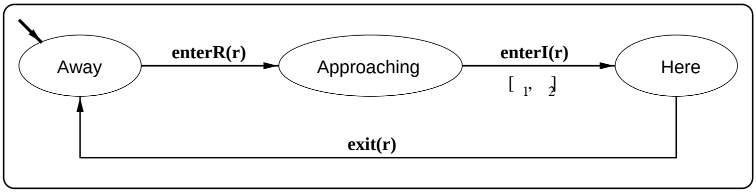
5.1 Problem Description

The GRC deals with the problem of controlling the movement of a gate which belongs to a railroad crossing with an arbitrary number of tracks (hence the predicate \generalized"). There is an obvious safety property, namely that whenever a train is inside the crossing, the gate has to be shut. A liveness condition ensures the utility of the system: if the crossing is empty after a train has passed, the gate must open within a certain time, unless there is another train approaching such that the gate could not stay open for some minimum amount of time.

5.2 A Formalization with Timed I/O Automata

As pointed out in Section 4.4, both the environment to be controlled and the controller are specified with timed I/O automata. The environment is modelled by two automata Trains and Gate, the controller by an automaton CompImpl. Putting these three automata in parallel we get the automaton SysImpl, which constitutes the behavior of the complete system.

To give an impression of what such specifications look like, we will take a closer look at the automaton Trains:



Since Trains has to account for an arbitrary number of tracks, both its state space and actions (all of which are output actions) are parameterized over a type of tracks which is not further specified. The graphical representation of Trains shows that on a track r, a train can be *away*, *approaching* or *here*. The action $enterR(r)$, with which a train passes from *away* to *approaching* will be an input action to the controller | think of a sensor positioned along track r in some distance from the crossing. Once a train is *approaching*, it can enter into the crossing with an action $enterI(r)$ | however there are two time-constraints given: the train may not enter before a time span of $''_1$ and not after a time span of $''_2$ has passed since it started to approach. Such time-constraints translate in a standard way into conditions under which a time-passage action can take place. We define the state space of Trains as a record type:

```
datatype location = Away j Approaching j Here
datatype MMTtime = 1 j btimec

record  tr-state =
   where ::     ) location
     rst ::  theAction ) MMTtime
   last ::  theAction ) MMTtime
```

Here rst and last map actions (theAction is a variant type which defines all occurring actions) into time bounds. These are expressed with a variant type, which either holds a time value (of a type time which is in the axiomatic type class timeD) or a value denoted with 1, signifying that no upper time bound exists; the operations , and the order relation $\leqslant$ of time extend in a natural way to MMTtime. Whenever a state is entered, from which a certain action has to occur within some given time bounds, then the components rst and last of this state are set accordingly; they are reset once the action in question takes place. The precondition for time-passage steps checks that time-passage does not invalidate one of the time bounds. The precondition of a constrained action checks the time bounds of this action.

The transition relation of the automaton Trains is specified using the mechanism of set comprehension of Isabelle/HOL. We only present a part of its formalization to give an idea what such a specification looks like:

```
defs  trains-trans =
      f(ss; a; tt):let s = content ss; s-now = now s;
                       t = content tt; t-now = now t in
        case a of
         ◁a⁰▷ ! (s-now = t-now)^
                case a⁰ of
                 enterR(r) ! if (where s)r = Away
                              then
                               t = s(where := (where s)(r := Approaching);
                                       rst := (( rst s)
                                              (enterI (r) := bs-now    1c))
                                      last := ((last s)
                                              (enterI (r) := bs-now    2c)))
                           else False
               j enterI (r) ! if (where s) r = Approaching
                              ^ ( rst s)(enterI (r))   bs-nowc
                              then
                               t = s(where := (where s)(r := Here);
                                       rst := (( rst s)
                                              (enterI (r) := b0c))
                                      last := ((last s)
                                              (enterI (r) := 1 )))
                          else False
               :::
         j ( t) ! if (8r: bs-now    tc   (last s)(enterI (r))
                  then
                   t-now = s-now    t ^ s = t  else False
      g
```

5.3 Informal Proof Outline

The correctness proof for the GRC as outlined informally in [9] proceeds as follows. The desired system behavior is formulated in terms of a projection of the admissible executions of an abstract automaton OpSpec. This automaton is built by first composing the automata Trains and Gate with an ‘empty’ controller, i.e. an automaton which has the same signature as CompImpl but puts no constraints onto any actions. To yield OpSpec, the resulting composition is then modified, restricting its possible behavior to the desired behavior. This is achieved by firstly adding new components to the state space, which we formalize like follows:

```
record  opSpec-state =
  system :: unit    tr-state   gate-state
  last1 :: MMTtime
  last2-up :: MMTtime
  last2-here :: MMTtime
```

Here system holds the state of the composite automaton (the ‘empty’ controller has only one single state, so unit can be used for its state-space). The three additional compontents of the state-space are then used to restrict the automaton’s behavior by adding a number of pre- and postconditions to the actions of the automaton. For example last1 is used as a deadline for some train to enter the crossing after the gate started to go down. An action *lower* of the gate automaton will set the deadline, whereas an action *enterI*(*r*) for any train *r* will reset it. last2-up and last2-here are used in a similar way to enforce further utility properties.

Correctness is established in two steps:

- The executions of the actual implementation are proven to be included in those of the automaton OpSpec under projection onto the environment.
- The executions of OpSpec in turn are shown to satisfy the axiomatic specification of the safety and utility property.

Only the first proof is described in some detail in [9]. Accordingly, we focus only on this proof part in our Isabelle formalization. Thus, the axiomatic specification is left out completely. The proof is carried out in [9] by first exhibiting a weak forward simulation between SysImpl and OpSpec. Then it is claimed that the desired inclusion of executions under projection on the component automata modeling the environment \follows from a general result about composition of timed automata"[9]. In contrast, we were able to use our formal theorem about execution inclusion as presented in Section 4.4.

5.4 The Formal Proof in HOLCF

About the Importance of Invariance Proofs. Showing invariants of automata seems to be the proof principle used most frequently when doing verification work within the theory of (timed) I/O automata. This is because firstly safety properties often are invariants, and secondly because invariants are used to factorize both invariance and simulation proofs: lemmas usually are formulated as invariants. Factorizing proofs is not only important for adding clarity, but in the context of interactive theorem proving also for keeping the intermediate proof states of a manageable size.

The practicability of invariants even suggests to prefer weak simulation relations in favour of forward simulations. In [10] it is shown that the notions of forward simulation and weak forward simulation are theoretically equivalent, i.e. whenever a forward simulation can be established, there exists also a weak forward simulation and vice versa. However, since for establishing the latter only *reachable* states of both automata have to be considered, invariants can be used to factor out information which otherwise would have to be made explicit in the simulation relation.

Setting the Stage for Semi-automated Invariance Proofs. Having established that invariance proofs may be regarded as the most important proof principle, it is clear that a high degree of automation for invariance proofs will be of considerable help. The guiding principle is that at least those parts of invariance proofs which in a paper proof would be considered as “obvious” should be handled automatically.

The proof rule for invariance proofs has been given in section 4.3. One has to show that an invariant holds for each start state, which usually is trivial, and that any transition from a reachable state for which the invariant holds leads again into a state which satisfies the invariant. This is done by making a case analysis over all the actions of an automaton. In a paper proof certain cases will be regarded as “trivial”, e.g. when a certain action will always invalidate the premise of the invariant to be shown.

It turned out that Isabelle’s generic simplifier and its classical reasoners [20, 21] would usually handle all the trivial cases and even more fully automatically when provided with information about the effect of an action in question. For automata like Trains this information is given directly by the definition, e.g:

$$\begin{aligned} \textbf{thm}\quad & s \xrightarrow{(t)}_{\mathsf{Trains}} t \Longrightarrow \\ & (\forall r.\ \lfloor \mathsf{now}\ s \quad t \rfloor \quad ((\mathsf{last}\ (\mathsf{content}\ s))(\mathit{enterI}\ (r)))) \\ & \wedge\ \mathsf{now}\ t = \mathsf{now}\ s \quad t\ \wedge\ \mathsf{content}\ s = \mathsf{content}\ t\ \wedge\ 0 < \quad t \end{aligned}$$

For composite automata, the combined effect of an action is easy to formulate. In the interactive proof however, care has to be taken to keep the proof state of moderate size. This can be achieved by using specially derived rules which from the definition of parallel composition between two automata derive the effect of parallel composition between more (in this case three) automata.

An invariance proof can now be carried out as follows. One starts with a special tactic which sets up an invariance proof by showing that the invariance holds for the start states and then performs a case analysis over the different actions. For each of these cases one first supplies information about the effect of the action in question with the respective lemma and then uses Isabelle’s simplifier and classical reasoner. This will often solve the case completely. Otherwise the resulting proof obligation will be simplified as far as possible for the user to continue by interactive proof.

Showing the Safety Property for SysImp. The safety property

$$\textbf{thm}\quad \mathsf{invariant\ SysImpl}\ (\ s.\ (\forall r.\ (\mathsf{where}'\ s\ r = \mathsf{Here} \Longrightarrow\ (\mathsf{setting}'\ s = \mathsf{Down}))))$$

where setting refers to the status of the gate (the primed identifiers have been defined as short-cuts for accessing the respective components of the automata that form SysImpl) has been established using five smaller and two larger lemmas — the latter correspond to lemmas 6.1 and 6.2 of [9]. Three of the five smaller lemmas were handled fully automatically, the other two would have been as well if the current version of Isabelle was better at arithmetic.

Showing the Weak Forward Simulation between SysImpl and OpSpec The definition of a weak forward simulation (see section 4.3) suggests to split the proof into three parts. The first two are trivial, but the third, at least in this case, turned out to be quite large. To achieve manageable interactive proof states, a lemma for each action has been proven, thus splitting the proof into smaller parts. Isabelle's simplifier and classical reasoner were of great help again; when a proof obligation could not be solved automatically but only simplified, usually either a guiding proof step was necessary, additional information was to be added in form of an invariant, or arithmetical reasoning was required. Apart from the safety property, five more invariants over SysImpl and two invariants over OpSpec were used. Only eight proof obligations have been assumed as axioms, being evidently true but very cumbersome to show within Isabelle because of the lacking support for arithmetical reasoning. One of them is for example to show that the following conjunction is contradictory:

$$\begin{aligned} & l_1 < l_2 \wedge t < l_1 \wedge l_3 < l_4 \\ & \quad \wedge l_4 = sch + {}_2 - {}_1 \wedge l_3 = l_2 + \quad + \\ & \quad \wedge sch \quad t + y + \quad \wedge \quad = y + \quad + {}_2 - {}_1 \end{aligned}$$

While it is cumbersome to show contradiction by hand in Isabelle, a tactic implementing a decision procedure for linear arithmetic could discharge this obligation immediately. All in all about one month was spent in formalizing the GRC.

6 Conclusions and Further Work

We have presented an Isabelle/HOLCF formalization of a solution to the GRC using timed I/O automata. Not only the correctness proof for the GRC, but also the underlying theory of timed I/O automata has been formalized. To our knowledge this is the first mechanized formalization both of substantial parts of the theory of timed IOA and of the simulation proof of the GRC as informally presented in [9]. Recognizing an inaccuracy in the meta-theory due to an incorrect use of the composition operator for timed automata in [9], we formulated and verified a proof principle for showing execution inclusion. This underlines the general advantages of fully formal tool-supported verification, as for example also observed in [5, 6, 15, 17].

Concerning related work, the TAME-project [3] which uses PVS for establishing a framework for specification and verification with timed I/O automata focuses on a standardized way to specify timed I/O automata and specialized tactics for reasoning about them. These specialized tactics are parameterized over theorems that are generated automatically from automata definitions. In effect this takes care of the preparation for semi-automated invariance proofs which we carried out by hand. Because of these tactics and the arithmetic decision procedures available in PVS, a higher degree of automation for invariance proofs is reached than in our work. However no meta-theoretical questions are treated | proof principles like simulation proofs have to be postulated. Hence new proof principles which might be discovered, just like the one for showing execution inclusion with a simulation proof used in the GRC, can only be added to the system with further postulates. Extending the system with proof principles in a formal way is impossible. Furthermore only the invariance proofs of the

GRC have been carried out within the TAME framework so far (see [2]). The simulation proof, which is a crucial part of the correctness proof, has not been covered.

Building upon a formalization of the theory of timed I/O automata, the experiences gathered from carrying out a case study may now be used to develop additional support for other verifications using timed I/O automata within Isabelle/HOLCF. Apart from the current lack of support for arithmetical reasoning, which of course for timed systems is quite a drawback, Isabelle's built-in solvers [20, 21] turned out to be quite fit for the task of automating invariance proofs. Here it would be helpful to have a tool which generates the proof scripts necessary for setting up the infrastructure for invariance proofs as explained in Section 5.4. Furthermore, the formalization of the theory of timed I/O automata should be extended to cover compositional reasoning. This would provide a sound basis for even larger and composed applications.

Acknowledgments

The first author would like to thank Tobias Nipkow for giving him the opportunity to visit the Isabelle group at TU München for a semester | the presented work has been carried out during this stay. Both authors are indebted to Tobias Nipkow for numerous discussions on the subject.

References

[1] The isabelle theories for timed i/o automata and the *Generalized Railroad Crossing*. available via `http://www.brics.dk/~grobauer/tioa/index.html`.

[2] Myla M. Archer and Constance L. Heitmeyer. Mechanical verification of timed automata: A case study. Technical Report NRL/MR/5546-98-8180, Naval Research Laboratory, 1998.

[3] Myla M. Archer, Constance L. Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proceedings of UITP '98*, July 1998.

[4] Stefan Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL. Master's thesis, TU München, 1998.

[5] Albert J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993{1004, 1990.

[6] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Proc. 2nd Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[7] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In *TPHOL'97, Proc. of the 10th International Workshop on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 89{104, 1997.

[8] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., December 1993. Extended abstract in Proceedings ICALP '94.

[9] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 1994.

[10] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations { Part II: timing based systems. Technical Report CS-R9314, CWI, 1993.

[11] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[12] Olaf Müller. I/O Automata and Beyond { Temporal Logic and Abstraction in Isabelle. In *TPHOL'98. Proc. of the 11th International Workshop on Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 331{348, 1998.

[13] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, TU München, 1998.

[14] Olaf Müller and Tobias Nipkow. Traces of I/O automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 580{594. Springer, 1997.

[15] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. To appear in Journal of Functional Programming.

[16] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics, Proceedings of TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, 1998.

[17] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe | definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161{170. ACM Press, New York, 1998.

[18] S. Owre, Rushby J., Shankar N., and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[19] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[20] Lawrence C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications*. MIT Press, 1997.

[21] Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. In *CADE-15 Workshop on Integration of Deductive Systems*, 1998.

[22] Markus Wenzel. Type classes and overloading in higher-order logic. In *Theorem Proving in Higher Order Logics, Proceedings of TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, 1997.

Formal Methods and Security Evaluation

Dominique Bolignano

Trusted Logic
Dominique.Bolignano@trusted-logic.fr

Formal methods have long been recognised as central to the development of secure system. Formal models of security policy and formal verification of cryptographic protocols have shown to be very useful to the development of real systems. But many new and promising research results in the area of security protocol verification, security architecture, or mobile code analysis, are still to be shown for practibility.

Quite recently the United States, United Kingdom, Germany, France, Canada, and the Netherlands released a jointly developed evaluation standard usually referred to as the "Common Criteria" (CC). This standard is to replace two earlier security standards: the american TCSEC and the european ITSEC.

For the highest levels these evaluation criteria require the use of formal methods during the earliest stages of the design and development of the security functions of a system. The use of formal methods is also considered in many cases as the best way to meet the semiformal requirement that are found at lower levels.

The new CC criteria tend to generalize the use of formal methods, with the goal of achieving a high level of assurance: formal methods are to be used at every stage of the "development" process (functional, internal, and interface specifications, plus high-level and low-level design). At the EAL5 level, which is the first level for wich formal methods have to be used, a formal model of the security policy has to be provided so as to verify the consistency of the policy. At EAL7 level which is the highest level of the scale, a formal description of security functions is required. Furthermore, the evaluator must be provided with a correspondence proof between the formal description of the security functions at a specific stage and their description at the subsequent stage.

The objective of the presentation is to investigate potential applications of formal methods in security evaluations. We will also describe current limitations as well as promising research directions.

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 291{291, 1999.

Importing MDG Verification Results into HOL

Haiyan Xiong[1], Paul Curzon[1], and Sofiène Tahar[2]

[1] School of Computing Science, Middlesex University, London, UK
{h.xiong, p.curzon}@mdx.ac.uk
[2] ECE Department, Concordia University, Montreal, Canada.
tahar@ece.concordia.ca

Abstract. Formal hardware verification systems can be split into two categories: theorem proving systems and automatic finite state machine based systems. Each approach has its own complementary advantages and disadvantages. In this paper, we consider the combination of two such systems: HOL (a theorem proving system) and MDG (an automatic system). As HOL hardware verification proofs are based on the hierarchical structure of the design, submodules can be verified using other systems such as MDG. However, the results of MDG are not in the appropriate form for this. We have proved a set of theorems that express how results proved using MDG can be converted into the form used in traditional HOL hardware verification.

1 Introduction

In general, machine-assisted hardware verification methods can be classified into two categories: interactive verification using a theorem prover and automated finite state machine (FSM) verification based on state enumeration. This study investigates the combination of two such systems: the HOL and MDG systems. The former is an interactive theorem proving system based on higher-order logic [8]. The latter is an automatic system based on Multiway Decision Graphs [2]. Tahar and Curzon [12] compared these two systems based on using both to independently verify real hardware: the Fairisle 4 by 4 switch fabric. Their results indicate that both systems are very effective and they are complementary. By combining them, it is hoped that the advantages of both can be obtained.

The MDG system is a hardware verification system based on Multiway Decision Graphs (MDGs). MDGs subsume the class of Bryant's Reduced Ordered Binary Decision Diagrams (ROBDD) [1] while accommodating abstract sorts and uninterpreted function symbols. The system combines a variety of different hardware verification applications implemented using MDGs [15]. The applications developed include: combinational verification, sequential verification, invariant checking and model checking.

The MDG verification approach is a black-box approach. During the verification, the user does not need to understand the internal structure of the design being verified. The strength of MDG is its speed and ease of use. However, it does not scale well to complex designs. In general, BDD based systems cannot

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 293–310, 1999.

cope with designs that combine datapaths and control hardware. MDG overcomes some of these problems. However, the largest example verified to date is the Fairisle 4 by 4 fabric [13].

In HOL, the specification language is higher-order logic. It allows functions and relations to be passed as arguments to other functions and relations. Higher-order logic is very flexible and has a well-defined and well-understood semantics. It also allows us to use a hierarchical verification methodology that effectively deals with the overall functionality of designs with complex datapaths. Designs that combine control hardware and datapaths can be verified. HOL scales better than MDG as illustrated by the fact that a 16 by 16 switch fabric constructed from elements similar to the 4 by 4 fabric has been verified in HOL [3] [4]. This is beyond the capabilities of MDG on its own. To complete a verification, however, a very deep understanding of the internal structure of the design is required, as it is a white-box approach. This enables the designer to gain greater insight into the system and thus achieve better designs. However, the learning curve is very steep and modeling and verifying a system is very time-consuming. The HOL system is generally better for higher-level reasoning in a more abstract domain.

Can we combine the two systems to reap the advantages of both? If we could, the problem size and complexity limits that can be handled in practice would be increased. We cannot, however, just accept that a piece of hardware verified using an automated verification tool such as the MDG system can be assumed correct in a HOL proof. In this paper, we focus on the theoretical underpinning of how to convert MDG results into HOL. In particular, we consider how to convert MDG results to appropriate HOL theorems as used in a traditional HOL hardware verification in the style of Gordon [6]. We give formalizations of MDG results in HOL based on the semantics of the MDG input language. We then suggest versions of these results that are of the form needed in a HOL hardware verification. Finally, we derive theorems that show that we can convert between these two forms. Thus, these theorems provide the specification for how MDG results can be imported into the HOL system in a useful form. This work is one step of a larger project to verify aspects of the MDG system in HOL so that MDG results can be trusted in the HOL system. The work presented here thus integrates with previous work to verify the MDG components library in HOL [5] and work to verify the MDG-HDL compiler.

Whilst this work concentrates on the MDG and HOL systems, the work has a much wider applicability. The theorems proved could be applicable for other verification systems with similar architectures based on reachability analysis or equivalence checking. Furthermore, the general approach taken is likely to be applicable to verification systems with different architectures.

The structure of this paper is as follows: in Section 2, we review related work. In Section 3, we overview the hierarchical hardware verification approach in HOL and motivate the need for MDG results to be in a particular form when importing them into the HOL system. In Section 4, we give the formal theorems that convert the MDG results into useful HOL theorems. These theorems have been verified using HOL. Our conclusions are presented in Section 5. Finally, ideas for further work are presented in Section 6.

2 Related Work

In 1993, Joyce and Seger [10] presented a hybrid verification system: HOL-Voss. In their system, several predicates were defined in the HOL system, which presents a mathematical link between the specification language of the Voss system (symbolic trajectory evaluation) and the specification language of the HOL system. A tactic VOSS_TAC was implemented as a remote function. It calls the Voss system that is then run as a child process of the HOL system. The Voss assertion can be expressed as a term of higher-order logic. Symbolic trajectory evaluation is used to decide whether or not the assertion is true. If it is true, then the assertion will be transformed into a HOL theorem and this theorem can be used by the HOL system to derive additional verification results. Zhu et al. [16] successfully applied HOL-Voss for the verification of the Tamarack-3 microprocessor.

Rajan et al. [11] proposed an approach for the integration of model checking with PVS: an automated proof checking system. The mu-calculus was used as a medium for communicating between PVS and a model checker. It was formalized by using the higher-order logic of PVS. The temporal operators that apply to arbitrary state spaces are given the customary fixpoint definitions using the mu-calculus. The mu-calculus expression was translated to an input that is acceptable by the model checker. This model checker was then used to verify the subgoals. In [9], a complicated communication protocol was verified by means of abstraction and model checking.

More recently, HOL98 has been integrated with the BuDDy BDD package [7]. HOL was used to formalize the QBF (Quantified Boolean Formulae) of BDDs. The formulae can be interactively simplified by using a higher-order rewriting tool such as the HOL simplifier to get simplified BDDs. A table was used to map the simplified formulae to BDDs. The BDD algorithms can also strengthen its deductive ability in this system.

In the work presented in this paper, we are not using the MDG system as an oracle to then prove results, already determined, by primitive inference in HOL, nor are we using HOL to improve the way MDG works. Furthermore, we are not just farming out general lemmas (e.g., propositional tautologies) that arise whilst verifying a particular hardware module and that can be proved more easily elsewhere. Our work is perhaps closer in spirit to that of the HOL-VOSS system than to other work in this sense. We are concerned with linking HOL to a dedicated hardware verification system that is in direct competition with it. It produces similar results about similar descriptions of circuits. We utilize this fact to allow MDG to be used when it would be easier than obtaining the result directly in HOL. The main contribution of this paper is that we present a methodology by which this can be done formally. We do not simply assume that the results proved by MDG are directly equivalent to the result that would have been proved in HOL.

3 Hierarchical Verification in a Combined System

In this section, we motivate the need for the results from a system such as MDG to be in a specific form by outlining the traditional HOL hierarchical hardware verification methodology. We also look at how an MDG result might be incorporated into such a verification approach.

Generally, when we use HOL to verify a design, the design is modeled as a hierarchy structure with modules divided into submodules as shown in Figure 1. The submodules are repeatedly subdivided until eventually the logic gate level is reached. Both the structure and behavior specifications of each module are given as relations in higher-order logic. The verification of each module is carried out by proving a theorem asserting that the implementation (its structure) implements (implies) the specification (its behavior). That is:

$$\vdash \mathit{implementation} \Rightarrow \mathit{specification} \qquad (1)$$

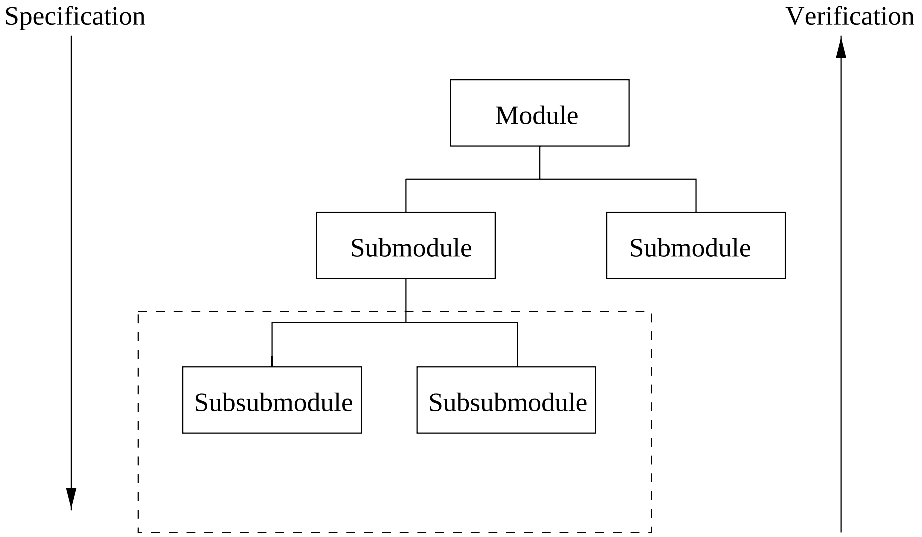


Fig. 1. Hierarchical Verification

The correctness theorem for each module states that its implementation down to the logic gate level satisfies the specification. The correctness theorem for each module can be established using the correctness theorems of its submodules. In this sense the submodule is treated as a black-box. A consequence of this is that different technologies can be used to address the correctness theorem for the submodules. In particular, we can use the MDG system instead of HOL to prove the correctness of submodules.

In order to do this, we need to formalize the results of the MDG verification applications in HOL. These formalizations have different forms for the different verification applications, i.e., combinational verification gives a theorem of one form, sequential verification gives a different form and so on. However, the most natural and obvious way to formalize the MDG results does not give theorems of the form that HOL needs if we are to use traditional HOL hardware verification techniques. We therefore need to be able to convert the MDG results into a form that can be used. In other words, we need to prove a series of translation theorems (one for combinational verification, one for sequential verification, etc.) that state how an MDG result can be converted to the traditional HOL form[1]:

$$\vdash \textit{Formalized MDG result} \quad (\textit{implementation} \quad \textit{specification}) \tag{2}$$

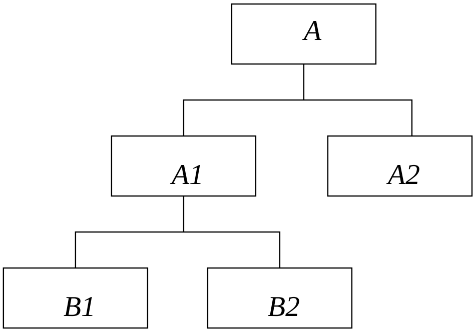


Fig. 2. The Hierarchy of Module A

To illustrate why we need a particular form of result in HOL consider the HOL verification of a system A. A theorem that the implementation satisfies its specification needs to be proved, i.e.

$$\vdash A\_imp \quad A\_spec \tag{3}$$

where A_imp and A_spec express the implementation and specification of system A, respectively. Suppose system A consists of two subsystems $A1$ and $A2$ and $A1$ is further subdivided as shown in Figure 2. The structural specification of A will be defined by the equation:

$$\vdash A\_imp = A1\_imp \wedge A2\_imp \tag{4}$$

[1] In this discussion, we have simplified the presentation for the purposes of exposition. In particular details of inputs and outputs are omitted.

where $A1\_imp$ is defined in a similar way. Thus (3) can be rewritten to

$$\vdash A1\_imp \wedge A2\_imp \supset A\_spec \tag{5}$$

The correctness theorem of the system A can be proved using the correctness statements about its subsystems. In other words, we independently prove the correctness theorems:

$$\vdash A1\_imp \supset A1\_spec \tag{6}$$

$$\vdash A2\_imp \supset A2\_spec \tag{7}$$

As these are implications, to prove (5) it is then sufficient to prove

$$\vdash A1\_spec \wedge A2\_spec \supset A\_spec \tag{8}$$

Thus we verify A by independently verifying its submodules, then treating them as black-boxes using the more abstract specification of $A1$ and $A2$ to verify A.

Suppose now that $A1$ was verified using MDG instead of HOL, but that we still wish to use the result in the verification of A. To make use of the result, we need MDG to also prove results of the form

$$\vdash A1\_imp \supset A1\_spec \tag{9}$$

so that the implementation can be substituted for a specification. However, results from MDG are not of this form[2]. For example, with sequential verification MDG proves a result about "reachable states" of a product machine. We need to show how such a result can be expressed as an implication about the actual hardware under consideration as above. If *A1_MDG_RESULT* is such a statement about a product machine, then we need to prove

$$\vdash \mathit{A1\_MDG\_RESULT} \supset (A1\_imp \supset A1\_spec) \tag{10}$$

Theorems such as this convert MDG results to the appropriate form to make the step between (5) and (8).

Ideally, we want a general theorem of this form that applies to any hardware verified using MDG's sequential verification tool. We also want similar results for the other MDG verification applications. In this paper, we prove such translation theorems for a series of MDG applications. This is described in the next section.

4 The Translation Theorems

In this section, we consider each of the verification applications of the MDG system in turn, describing the conversion theorem required to convert results to a form useful within a HOL proof. Each of these theorems has been proved within the HOL system.

[2] We give details of the form of theorems that MDG does prove in the next section.

4.1 Combinational Verification of Logic Circuits

The simplest verification application of MDG is the checking of equivalence of input-output for two combinational circuits. A combinational circuit is a digital circuit without state-holding elements or feedback loops, so the output is a function of the current input. The MDGs representing the input-output relation of each circuit are computed by a relational product algorithm to form the MDGs of the components of the circuit. Because an MDG is a canonical representation, we can check whether the two MDGs are isomorphic and so the circuits are equivalent. It is simple to formalize this in HOL. We use $M(ip; op)$ and $M^0(ip; op)$ to represent the circuits (machines) being compared. M is a relation on input traces (given by ip) and output traces (given by op). The relation is true if op represents a possible output trace for the given input trace ip and is false otherwise. M^0 is a similar relation on inputs (ip) and outputs (op). An MDG combinational verification result can be formalized as:

$$‘\ 8\ ip\ op{:}\ M\ (ip;\ op)\ =\ M^0\ (ip;\ op) \qquad (11)$$

It verifies that the two circuits are identical in behavior for all inputs and outputs. If ip and op are possible input and output traces for M, then they are also possible traces for M^0, and vice versa. This is not in the form of an implication as described above. However, the MDG result does not need to be converted to a different form for it to be useful in a HOL hardware verification, since an equality can be used just as well as an implication.

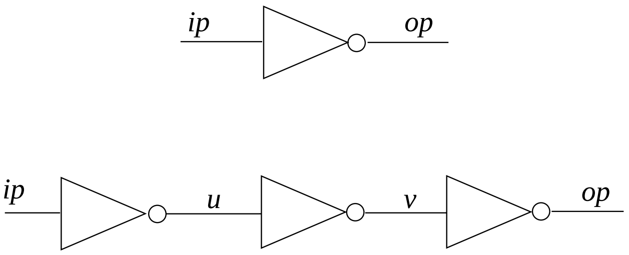


Fig. 3. Two Equivalent Combinational Circuits

Example 1. Consider the two circuits shown in Figure 3. Assume they have been verified to be equivalent using MDG combinational equivalence checking. We will show in the following how to convert the MDG result to a useful HOL theorem.

The first circuit is a single NOT gate that can be specified as:

$$‘\ 8\ in\ op{:}\ \mathrm{NOT}\ (ip;\ op)\ =\ (8\ t{:}\ op\ t\ =\ \ \ ip\ t)$$

The second circuit consists of three NOT gates in series and can be formalized as:

$$\text{‘ } 8\ ip\ op\colon\ \text{NOT3}\ (ip;\ op) \ = $$
$$9\ u\ v\colon\ \text{NOT}\ (ip;\ u)\ \hat{}\ \text{NOT}\ (u;\ v)\ \hat{}\ \text{NOT}\ (v;\ op)$$

The MDG verification result can be stated as

$$\text{‘ } 8\ ip\ op\colon\ \text{NOT3}\ (ip;\ op) \ = \text{NOT}\ (ip;\ op)$$

This theorem has the form that we need in the HOL verification system, so no translation theorem is required

4.2 Combinational Verification of Sequential Circuits

Combinational verification can also be used to compare two sequential circuits when a one-to-one correspondence between their registers exists and is known. In this situation M and M^0 are relations on inputs (ip), outputs (op) and states (s). The result of the MDG proof can then be stated as:

$$\text{‘ } 8\ ip\ op\ s\colon\ M\ \ (ip;\ op;\ s)\ \ =\ \ M^0\ (ip;\ op;\ s) \qquad (12)$$

This is explicitly concerned with state in the form of the variable s. In a HOL verification the way we model hardware by a relation between input and output traces means that we do not, in general, need to model the state explicitly. Traces are described as history functions giving the value output at each time instance. A register can then, for example, be specified as

$$\text{‘ REGH}\ (ip;\ op) = (op\ 0 = \text{F})\ \hat{}\ (8\ t\colon (op\ (t+1) = ip\ t)) \qquad (13)$$

There is no explicit notion of state in this definition | we just refer to values at an earlier time instance.

MDG descriptions in MDG-HDL on the other hand explicitly include state: state variables are declared and state transition functions given. The MDG version could be formalized in HOL by

$$\text{‘ REGM}\ (ip;op;s) = \text{INIT}\ s\ \hat{}\ \text{DELTA}\ (ip;s)\ \hat{}\ \text{OUT}\ (ip;op;s) \qquad (14)$$

where

$$\text{INIT}\ s = (s\ 0 = \text{F})$$
$$\text{DELTA}\ (ip;\ s) = (8\ t\colon s\ (t+1) = ip\ t)$$
$$\text{OUT}\ (ip;\ op;\ s) = (8\ t\colon op\ t = s\ t)$$

We therefore need a way of abstracting away this state when converting to the HOL form. As was explained in Section 3, ultimately the correctness theorem that HOL wants should have the form of (1). We can hide the state using existential quantification and obtain:

$$\text{‘ } 8\ ip\ op\colon (9\ s\colon M\ \ (ip;\ op;\ s)) \qquad (9\ s\colon M^0\ (ip;\ op;\ s)) \qquad (15)$$

This is of the required form:

$$M\_imp\ (ip;\ op) \quad M\_spec\ (ip;\ op)$$

where

$$M\_imp\ (ip;\ op) = (9\ s{:}\ M\ (ip;\ op;\ s))$$
$$M\_spec\ (ip;\ op) = (9\ s{:}\ M^0\ (ip;\ op;\ s))$$

In this situation the converting theorem is:

$$‘\ 8\ M\ M^0{:}$$
$$(8\ ip\ op\ s{:}\ M\ (ip;\ op;\ s) = M^0\ (ip;\ op;\ s))$$
$$(8\ ip\ op{:}\ (9\ s{:}\ M\ (ip;\ op;\ s)) \quad (9\ s{:}\ M^0\ (ip;\ op;\ s))) \qquad (16)$$

We have proved this theorem in HOL. Note that the relations M and M^0 are universally quantified variables. The theorem thus applies to any hardware for which an MDG result is verified.

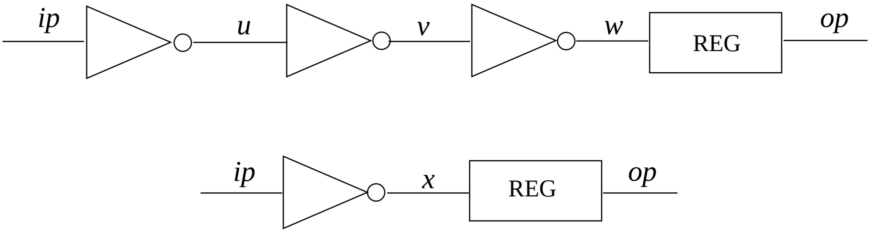


Fig. 4. Two Equivalent Sequential Circuits

Example 2. Consider verifying the sequential circuits in Figure 4 using combinational equivalence. We check that three not gates and a register are equivalent to a single not gate and register. We use REGNOT3M to formalize the first circuit,

$$‘\ \text{REGNOT3M}(ip;\ op;\ s) =$$
$$9\ u\ v\ w{:}$$
$$\text{NOT}\ (ip;\ u)\ \hat{}\ \text{NOT}\ (u;\ v)\ \hat{}\ \text{NOT}\ (v;\ w)\ \hat{}\ \text{REGM}\ (w;\ op;\ s)$$

We use REGNOTM to formalize the second circuit,

$$‘\ \text{REGNOTM}\ (ip;\ op;\ s) = 9\ x{:}\ \text{NOT}\ (ip;\ x)\ \hat{}\ \text{REGM}\ (x;\ op;\ s)$$

Suppose we have verified that these two circuits are equivalent using the MDG system. The MDG verification result can be stated as:

$$‘\ 8\ ip\ op\ s{:}\ \text{REGNOT3M}\ (ip;\ op;\ s) = \text{REGNOTM}\ (ip;\ op;\ s)$$

Combining this with our conversion theorem (16), we obtain

$$\vdash \forall ip\ op.$$
$$(\exists s.\ \text{REGNOT3M}\ (ip, op, s)) \qquad (\exists s.\ \text{REGNOTM}\ (ip, op, s)) \quad (17)$$

This is not quite the theorem we would have proved if the verification was done directly in HOL. However, it can be obtained if we first prove the theorems:

$$\vdash \text{REGNOT3H}\ (ip, op) = (\exists s.\ \text{REGNOT3M}\ (ip, op, s)) \quad (18)$$
$$\vdash \text{REGNOTH}\ (ip, op) = (\exists s.\ \text{REGNOTM}\ (ip, op, s)) \quad (19)$$

where REGNOT3H and REGNOTH are stateless HOL descriptions of the corresponding circuits[3]. They are defined as follows:

$$\vdash \text{REGNOT3H}\ (ip, op) =$$
$$\exists u\ v\ w.$$
$$\text{NOT}\ (ip, u) \wedge \text{NOT}\ (u, v) \wedge \text{NOT}\ (v, w) \wedge \text{REGH}\ (w, op)$$
$$\vdash \text{REGNOTH}\ (ip, op) =$$
$$\exists x.\ \text{NOT}\ (ip, x) \wedge \text{REGH}\ (x, op)$$

Finally, using (18) and (19) to rewrite (17), we obtain the theorem which is needed in a traditional HOL verification.

$$\vdash \forall ip\ op.\ \text{REGNOT3H}\ (ip, op) \qquad \text{REGNOTH}\ (ip, op)$$

It should be noted that the actual verification application of MDG does not do state traversal so the state is not actually used in the MDG verification process. However the MDG hardware description language (HDL) is still used as the description language. Therefore the explicit introduction of state is required if the relations are to represent semantic objects of MDG-HDL. This is of importance in our work since we ultimately intend to link these theorems with ones that explicitly refer to the semantics of MDG-HDL.

4.3 Sequential Verification

The behavioral equivalence of two abstract state machines (Figure 5) is verified by checking that the machines produce the same sequence of outputs for every sequence of inputs. The same inputs are fed to the two machines M and M' and then reachability analysis is performed on their product machine using an invariant asserting the equality of the corresponding outputs in all reachable states. This effectively introduces new “hardware” (see Figure 5) which we refer to here as PSEQ (the Product machine for SEQuential verification). PSEQ has the same inputs as M and M', but has as output a single Boolean signal ($flag$). The outputs op and op' of M and M' are input into an equality checker. On each cycle, PSEQ outputs true if op and op' are identical at that time, and false otherwise. PSEQ can be formalized as

[3] The need for such theorems will be discussed further in Section 6.

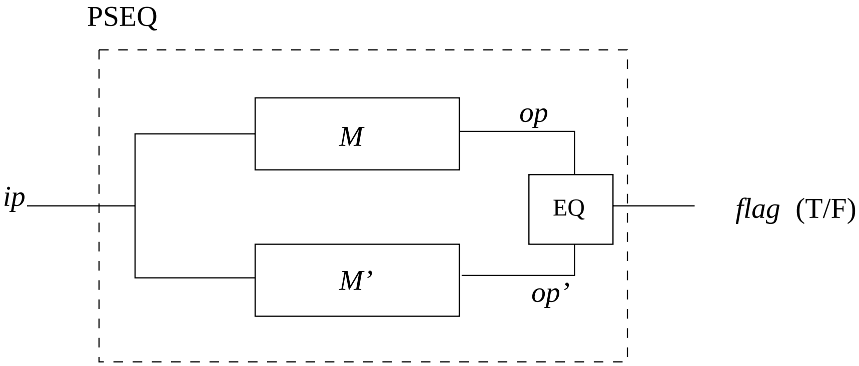


Fig. 5. The Product Machine used in MDG Sequential Verification

$$\vdash \text{PSEQ}\ (ip, flag, op, op', s, s', M, M') = M\ (ip, op, s) \wedge M'\ (ip, op', s') \wedge \text{EQ}\ (op, op', flag) \qquad (20)$$

where EQ is the equality checker defined as:

$$\vdash \text{EQ}\ (op, op', flag) = (\forall t.\ flag\ t = (op\ t = op'\ t)) \qquad (21)$$

The result that MDG proves about PSEQ is that the flag output is always true, i.e., the outputs are equal for all inputs. This can be formalized as

$$\vdash \forall\ s\ s'\ ip\ op\ op'.\ \text{PSEQ}\ (ip, flag, op, op', s, s', M, M') \supset (\forall t.\ flag\ t = \text{T}) \qquad (22)$$

Note that this is not of the form $P\_imp \supset P\_spec$, (i.e., implementation implies specification) for M and M' but is of that form for the fictitious hardware PSEQ. To make use of such a result in a HOL hardware verification, we need to convert it to that form for M and M'. This can be done in a series of steps starting from (22). Expanding the definitions and rewriting with the value of flag, we obtain

$$\vdash \forall\ s\ s'\ ip\ op\ op'.\ M\ (ip, op, s) \wedge M'\ (ip, op', s') \supset (\forall t.\ op\ t = op'\ t) \qquad (23)$$

i.e., we have proved a lemma:

$$\begin{aligned}\vdash\ & \forall M\ M'.\\ & (\forall s\ s'\ ip\ op\ op'.\\ &\quad \text{PSEQ}\ (ip, flag, op, op', s, s', M, M') \supset \forall t.\ flag\ t = \text{T}) \supset\\ & (\forall s\ s'\ ip\ op\ op'.\ M\ (ip, op, s) \wedge M'\ (ip, op', s') \supset (\forall t.\ op\ t = op'\ t)) \qquad (24)\end{aligned}$$

This is still not in an appropriate form, however. We need to abstract away from the states as with combinational verification. The theorem should also be in the form of (1). The machine M can be considered as the structure specification (implementation) and machine M' the behavior specification (specification). Based on this consideration, the theorem that HOL needs is as follows:

$$\vdash \forall ip\ op.\ (\exists s.\ M\ (ip, op, s)) \Rightarrow (\exists s'.\ M'\ (ip, op, s')) \tag{25}$$

i.e., for all input and output traces if there exists a reachable sequence of states s that satisfy the relation $M\ (ip, op, s)$, then must exist a reachable state s' that satisfies the relation $M'\ (ip, op, s')$. As mentioned above, the converting theorem from MDG to HOL should be in the form of (2). For sequential verification the conversion theorem should be

$$(22) \Rightarrow (25).$$

To prove this, given (24) it is sufficient to prove

$$(23) \Rightarrow (25).$$

However, this can only be proved with an additional assumption. Namely, for all possible input traces, the behavior specification M' can be satisfied for some output and state traces (i.e., there exists at least one output and state trace for which the relation is true):

$$\vdash \forall ip.\ \exists op'\ s'.\ M'\ (ip, op', s') \tag{26}$$

This means that the machine must be able to respond whatever inputs are given. This should always be true for reasonable hardware. You should not be able to give inputs which break it. For any input sequence given to this machine, at least one output and state sequence will correspond. Therefore, we can actually only prove $\vdash (22) \wedge (26) \Rightarrow (25)$,

$$\begin{array}{l}
\vdash \forall\ M\ M'.\\
\quad ((\forall s\ s'\ ip\ op\ op'.\\
\qquad \text{PSEQ}\ (ip, flag, op, op', s, s', M, M') \Rightarrow \forall t.\ flag\ t = \text{T}) \wedge\\
\quad (\forall ip.\ \exists op'\ s'.\ M'\ (ip, op', s')))\\
\qquad\quad \Rightarrow (\forall ip\ op.\ (\exists s.\ M\ (ip, op, s)) \Rightarrow (\exists s'.\ M'\ (ip, op, s')))
\end{array} \tag{27}$$

With the same reasoning, the machine M' could have been considered as the structural specification and machine M could have been considered as the behavioral specification. We would then need the assumption

$$\vdash \forall ip.\ \exists op\ s.\ M\ (ip, op, s) \tag{28}$$

We would obtain the alternative conversion theorem (29)

$$\begin{array}{l}
\vdash \forall\ M\ M'.\\
\quad ((\forall s\ s'\ ip\ op\ op'.\\
\qquad \text{PSEQ}\ (ip, flag, op, op', s, s', M, M') \Rightarrow \forall t.\ flag\ t = \text{T}) \wedge\\
\quad (\forall ip.\ \exists op\ s.\ M\ (ip, op, s)))\\
\qquad\quad \Rightarrow (\forall ip\ op.\ (\exists s'.\ M'\ (ip, op, s')) \Rightarrow (\exists s.\ M\ (ip, op, s)))
\end{array} \tag{29}$$

Both these theorems have been verified in HOL. As with combinational verification, the universal quantification of M and M' means the theorems can be instantiated for any hardware under consideration. The symmetry in these equations is as might be expected given the symmetry of PSEQ.

Example 3. The circuits given in Figure 4 can also be verified using sequential verification. We shall show how to convert the result obtained to form a useful HOL theorem.

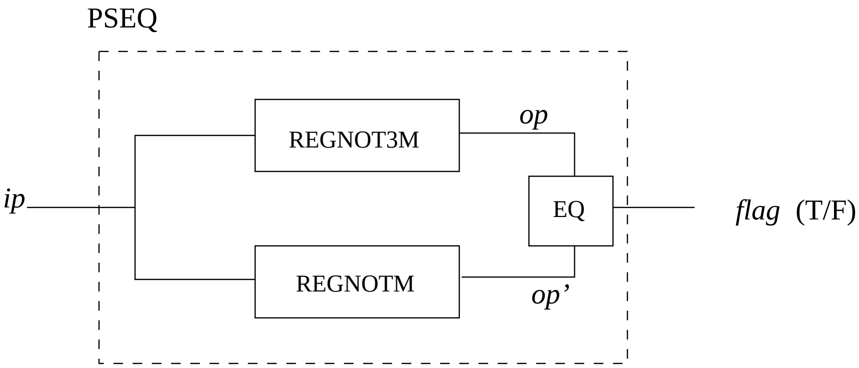


Fig. 6. The Machine used for Sequential Verification of the REGNOT3M Circuit

The MDG verification result can be stated as

$$\begin{array}{l}\vdash \forall ip\ op\ s.\\ \quad \mathrm{REGNOT3M}\ (ip, op, s) \wedge \mathrm{REGNOTM}\ (ip, op, s) \wedge \mathrm{EQ}\ (op, op', flag)\\ \qquad\qquad (\forall t.\ flag\ t = \mathrm{T})\end{array}$$

We have proved the required theorem that states that the REGNOTM unit responds whatever inputs are given.

$$\vdash \forall ip.\ (\exists op'\ s'.\ \mathrm{REGNOTM}\ (ip, op', s'))$$

Combining the above two theorems with our conversion theorem (27), we obtain:

$$\begin{array}{l}\vdash\ \forall ip\ op.\\ \quad \exists s.\ \mathrm{REGNOT3M}\ (ip, op, s) \quad \exists s'.\ \mathrm{REGNOTM}\ (ip, op, s')\end{array} \qquad (30)$$

Finally, after using (18) and (19) to rewrite (30), we obtain a theorem in a form that can be used in a HOL verification.

$$\vdash\ \forall ip\ op.\ \mathrm{REGNOT3H}\ (ip, op) \quad \mathrm{REGNOTH}\ (ip, op)$$

4.4 Invariant Checking

Systems such as MDG also provide property/invariant checking. Invariant checking is used for verifying that a design satisfies some specific requirements. This is useful since it gives the designer confidence at low verification cost. In MDG, reachability analysis is used to explore and check that a given invariant (property) holds in all the reachable states of the sequential circuit under consideration, M. We consider one general form of property checking here.

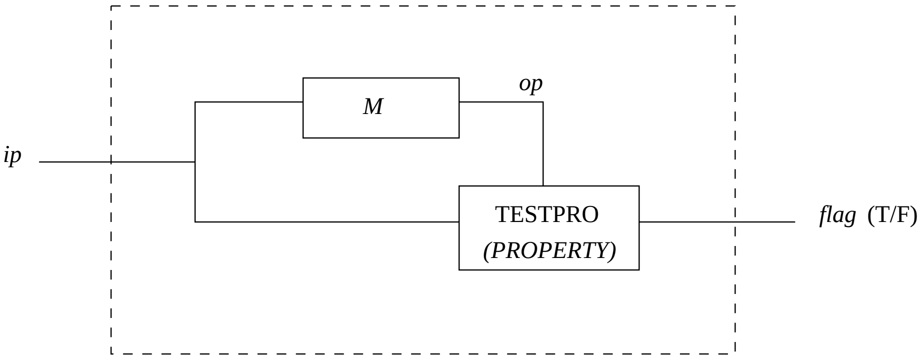


Fig. 7. The Machine Verified in Invariant Checking

As was the case for sequential verification, we introduce new \hardware" (see Figure 7) which we refer to as PINV (Product machine for INVariant checking). It consists of the original hardware and hardware representing the test property[4] wired together so that the property circuit has access to both the inputs and outputs of the circuit under test. PINV checks whether the outputs of the machine M satisfy the specific property or not. It is formalized as follows:

$$\begin{aligned}\vdash \text{PINV}\ (ip;\ flag;\ op;\ s;\ M;\ PROPERTY) &= \\ M\ (ip;\ op;\ s) \wedge \text{TESTPRO}\ (ip;\ op;\ flag;\ PROPERTY)\end{aligned} \tag{31}$$

where

$$\begin{aligned}\vdash \text{TESTPRO}\ (ip;\ op;\ flag;\ PROPERTY) &= \\ (\forall t.\ flag\ t = PROPERTY\ (ip\ t;\ op\ t))\end{aligned} \tag{32}$$

i.e., TESTPRO is a piece of hardware which tests if its inputs and outputs satisfy some specific requirements given at each time instance by *PROPERTY*. *PROPERTY* is a relation on input and output values. Again in discussing correctness

[4] Invariants in MDG must be written in or converted to the same hardware description language as the actual hardware.

it is actually a result about this different hardware that we obtain from the property checking. The result that the property checking proves about PINV can be stated as:

$$\begin{array}{l}\vdash \forall\, ip\ s\ op\ M\ PROPERTY.\\ \qquad \mathrm{PINV}\,(ip, flag, op, s, M, PROPERTY) \supset \forall t.\ flag\ t = \mathrm{T} \qquad (33)\end{array}$$

i.e., its specification is that the $flag$ output should always be true. Note that this is not of the form (1) (i.e., implementation implies specification) for M but in that form for the fictitious hardware PINV. To make use of such a result in a HOL hardware verification we need to convert it to the form:

$$\vdash \forall\, ip\ op.\ \exists s.\ M\,(ip, op, s) \supset \forall t.\ PROPERTY\,(ip\ t, op\ t) \qquad (34)$$

i.e., for all input and output sequences, if there exists a reachable state trace, s, satisfying the relation $M\,(ip, op, s)$ then the relation $PROPERTY$ must be true for the input and output values at all times. In other words, the machine M satisfies the specific requirement $\forall t.\ PROPERTY\,(ip\ t, op\ t)$. Hence the conversion theorem for invariant checking is:

$$\begin{array}{l}\vdash \forall\, M\ PROPERTY.\\ \quad (\forall\, ip\ op\ s.\\ \quad\ (\mathrm{PINV}\,(ip, flag, op, s, M, PROPERTY) \supset \forall t.\ flag\ t = \mathrm{T}))\\ \qquad \supset (\forall\, ip\ op.\ \exists s.\ M\,(ip, s, op) \supset \forall t.\ PROPERTY\,(ip\ t, op\ t)) \qquad (35)\end{array}$$

We have proved this general conversion theorem in HOL. Once more the theorems can be instantiated for any hardware and property under consideration.

5 Conclusions

We have formally specified the correctness results produced by four different hardware verification applications using HOL. We have in each case proved a theorem that translates them into a form usable in a traditional HOL hardware verification, i.e., that the structural specification implements the behavioral specification. The first applications considered were the checking of input-output equivalence of two combinational circuits and the similar comparison of two sequential circuits when a one-to-one correspondence between their registers exists and is known. The next application considered was sequential verification, which checks that two abstract state machines produce the same sequence of outputs for every sequence of inputs. Finally, we considered a general form of the checking of invariant properties of a circuit.

The verification applications considered were based on those of the MDG hardware verification system. We have thus given a theoretical basis for converting MDG results into HOL. Furthermore, by proving these theorems in HOL itself we have given practical tools that can be used when verifying hardware using a combined system | MDG results can be initially imported into HOL as theorems in the MDG form and converted to the appropriate HOL form using the conversion theorems. This gives greater security than importing the theorems directly in the HOL form, as mistakes are less likely to be introduced. Alternatively, if theorems of the HOL form are created directly, then the conversion theorems provide the specification of the software that actually creates the imported theorem.

Whilst the verification applications were based on the MDG system and the proof done in HOL, the general approach could be applied to the importing of results between other systems. The results could also be extended to other verification applications. Furthermore, our treatment has been very general. The theorems proved do not explicitly deal with the MDG-HDL semantics or multiway decision graphs. Rather they are given in terms of general relations on inputs and outputs. Thus they are applicable to other verification systems with a similar architecture based on reachability analysis, equivalence checking and/or invariant checking.

The translation theorems are relatively simple to prove. The contribution of the paper is not so much in the proofs of the theorems, but in the methodology of using imported results presented. It is very ease to fall into the trap of assuming that because a result has been obtained in one system, an \obviously" corresponding result can be asserted in another. An example of the dangers is given by the extra assumption needed for sequential verification and invariant checking that the circuit verified can respond to any possible input. It could easily be overlooked. By formalizing the results in the most natural form of the verification application, and proving it equivalent to the desired form, we reduce the chances of such problems occurring.

6 Discussion and Further Work

The reason that the state must be made explicit in the formalism of MDG is that the semantics of MDG-HDL | the input language to the MDG system | has an explicit notion of state. We have used relations M and M^0 to represent MDG semantic objects. Ultimately we intend to combine the theorems described here with correctness theorems about the MDG-HDL compiler which translates MDG-HDL programs into decision graphs [5] [14]. This will provide a formal link between the low level objects actually manipulated by the verification system (and about which the verification results really refer) and the results used in subsequent HOL proofs. Compiler verification involves proving that the compiler preserves the semantics of all legal source programs. It thus requires the definition of both a syntax and semantics of HDL programs. Using an explicit notion of state in the translation theorems ensures they will be compatible with the semantics of the MDG-HDL language used in the compiler verification.

However, if proving results directly in HOL, as we saw, introducing such an explicit notion of state is unnecessary. We could do so for the convenience of combining systems. However, this would make subsequent specification and verification in HOL more cumbersome. Consequently, we hide the state in the conversion theorems, introducing terms such as:

$$\exists s.\ M(ip, op, s)$$

However, this implies that we also define HOL components in this way. For example, it suggests a register is defined as

$$\vdash \forall ip\ op.\ \mathrm{REGH}\ (ip, op) = \exists s.\ \mathrm{REGM}\ (ip, op, s) \tag{36}$$

where REGM is as defined in (14). We actually want to give and use definitions as in (13), however, and derive (36). As a consequence for any component verified in MDG we must prove a theorem that the two versions are equivalent as we did in Examples 2 and 3.

In general, we need to prove theorems of this form for each MDG-HDL basic component. We then need to construct a similar theorem for the whole circuit whose verification result is to be imported. Such proofs can be constructed from the theorems about individual components. For instance, we must prove for any network of components (n) that

$$\vdash \forall n.\ \mathrm{HOL\_DESCRIPTION}\ n\ (ip, op) = \\ \exists s.\ \mathrm{MDG\_DESCRIPTION}\ n\ (ip, op, s).$$

If we prove this theorem, we can then convert our importing theorems into ones without state automatically. To do such a proof requires a syntax of circuits: precisely what is needed in the verification of the MDG compiler as noted above.

Acknowledgments

We are grateful to Prof. Ann Blandford for her helpful comments and suggestions. This work is funded by EPSRC grant GR/M4522, and a studentship from the School of Computing Science, Middlesex University. Travel funding was provided by the British Council, Canada.

References

1. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, 24(3), September 1992.
2. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
3. P. Curzon. The formal verification of the Fairisle ATM switching element. Technical Report 329, University of Cambridge, Computer Laboratory, March 1994.
4. P. Curzon. Tracking design changes with formal machine-checked proof. *The Computer Journal*, 38(2), July 1995.
5. P. Curzon, S. Tahar, and O. Aït-Mohamed. The verification of aspects of a decision diagram based verification system. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher-Order Logics: Emerging Trends*, pages 31–46. Department of Computer Science, The Australian National University, 1998.

6. M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: the 1985 Edinburgh Workshop on VLSI*, pages 153{177. North-Holland, 1986.
7. M. J. C. Gordon. Combining deductive theorem proving with symbolic state enumeration. Presented at *21 Years of Hardware Verification*, Royal Society Workshop to mark 21 years of BCS FACS, http://www.cl.cam.ac.uk/users/mjcg/BDD, December 1998.
8. M. J. C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.
9. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662{682, March 1996.
10. J. Joyce and C. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *the 30th Design Automation Conference*, 1993.
11. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification*, number 939 in Lecture Notes in Computer Science, pages 84{97. Springer-Verlag, 1995.
12. S. Tahar and P. Curzon. Comparing HOL and MDG: A case study on the verification of an ATM switch fabric. To appear in the *Nordic Journal of Computing.*
13. S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Aït-Mohamed. Modeling and automatic formal verification of the Fairisle ATM switch fabric using MDGs. To appear in *IEEE Transactions on CAD of Integrated Circuits and Systems.*
14. H. Xiong and P. Curzon. The verification of a translator for MDG's components in HOL. In *MUCORT98, Third Middlesex University Conference on Research in Technology*, pages 55{59, April 1998.
15. Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual.* University of Montreal, Dept. D'IRO, 1996.
16. Z. Zhu, J. Joyce, and C. Seger. Verification of the Tamarack-3 microprocessor in a hybrid verification environment. In *Higher-Order Logic theorem proving and Its Applications, The 6th International Workshop*, number 780 in Lecture Notes in Computer Science, pages 252{266. B. C., Canada, August 1993.

Integrating Gandalf and HOL

Joe Hurd*

Computer Laboratory
University of Cambridge
joe.hurd@cl.cam.ac.uk

Abstract. Gandalf is a first-order resolution theorem-prover, optimized for speed and specializing in manipulations of large clauses. In this paper I describe GANDALF_TAC, a HOL tactic that proves goals by calling Gandalf and mirroring the resulting proofs in HOL. This call can occur over a network, and a Gandalf server may be set up servicing multiple HOL clients. In addition, the translation of the Gandalf proof into HOL fits in with the LCF model and guarantees logical consistency.

1 Introduction

Gandalf [8] [9] [10] is a resolution theorem-prover for first-order classical logic with equality. It was written in 1994 by Tanel Tammet (tammet@cs.chalmers.se) and won the annual CASC competitions in 1997 and 1998, beating off competition from Spass, Setheo and Otter. Gandalf is optimized for speed, and specialises in manipulations of large clauses.

HOL [3] [7] is a theorem-prover for higher-order logic, with a small logical core to ensure consistency and a highly general meta-language in which to write proof procedures.

In this paper I describe GANDALF_TAC, a HOL tactic that sits between these two provers, enabling first-order HOL goals to be proved by Gandalf. Using a first-order prover within a higher-order logic is not new, and many ideas have been explored here before (e.g., FAUST and HOL [6], SEDUCT and LAMBDA [2], 3TAP and KIV [1]). However, there are two significant novelties in the work presented here:

- The use of a completely seperate 'off-the-shelf' theorem-prover, treating it as a black box.
- The systematic use of a generic plug-in interface.

There is an increasing trend for HOL tactics to perform proof-search as much as possible outside the logic, for reasons of speed. When a proof is found, only then is the complete verification executed in the logical core, producing the official theorem (and incidentally validifying the correctness of the proof search). Perhaps the first instance of a first-order prover regenerating its proof in HOL

* Supported by an EPSRC studentship

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 311–321, 1999.

was the FAUST prover developed at Karlsruhe[6], and more recently John Harrison wrote MESON_TAC [4]; a model-elimination prover for HOL that performs the search in ML. GANDALF_TAC extends this idea, completely seperating the proof search from the logical core by sending it to an external program (which was probably not designed with this application in mind).

GANDALF_TAC is a Prosper plug-in, and as such does not purport to be a universal proof procedure, but rather a component of an underlying proof infrastructure. The Prosper[1] project aims to deliver the benefits of mechanised formal analysis to system designers in industry. Essential to this goal is an open proof architecture, allowing formal methods technology to be combined in a modular fashion. To this end the Prosper plug-in interface[2] was written by Michael Norrish, enabling developers to add specialised verification tools (like Gandalf) to the core proof engine in a relatively uniform way. GANDALF_TAC was the first plug-in to be written, and in a small way provided a test of concept of the Prosper frame-work.

Although it is a digression from the main point of the paper, since nothing has been published on Prosper to date it might be appropriate to provide an short description of the system. Fig. 1 shows an overview of the open proof architecture, in which client applications submit requests to a Core Proof Engine (CPE) server, which in turn might farm out subproblems to plug-in servers. The Plug-In Interface (PII) exists on both the CPE side as an ML API, and on the plug-in side as an internet server that spawns the desired plug-in on request. GANDALF_TAC is implemented in ML and communicates with a Gandalf wrapper

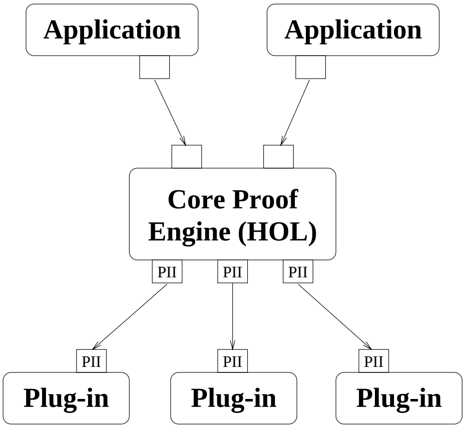


Fig. 1. Overview of the Prosper architecture.

[1] The Prosper homepage is at http://www.dcs.gla.ac.uk/prosper/.

[2] The Prosper plug-in interface homepage is at http://www.cl.cam.ac.uk/users/-mn200/prosper.

script by passing strings to and fro. The wrapper script takes the input string, saves it to a file, and invokes Gandalf with the filename as a command-line parameter, passing back all output. This is illustrated in Fig. 2.

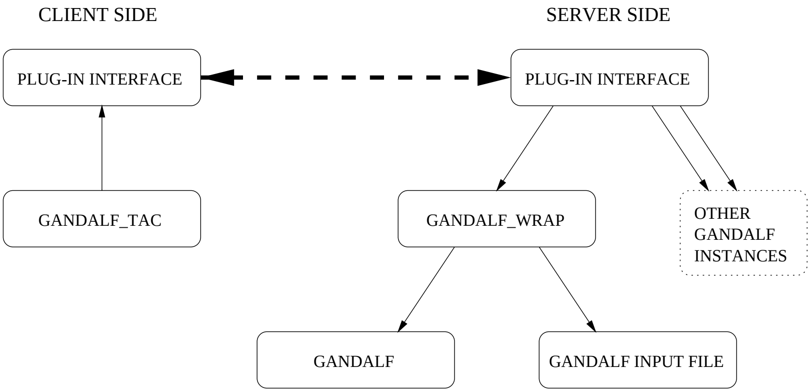


Fig. 2. The Gandalf internet server.

2 How It Works

Briefly, GANDALF_TAC takes the input goal, converts it to a normal form, writes it in an acceptable format, sends the string to Gandalf, parses the Gandalf proof, translates it to a HOL proof, and proves the original goal. Fig. 3 shows the procedure in pictorial form.

We will run through each stage in turn, tracking the metamorphosis of the goal

$$\forall a b.\ \exists x.\ P a \lor P b \Rightarrow P x$$

2.1 Initial Primitive Steps

In the first stage of processing we assume the negation of the goal:

$$\{\neg(\forall a b.\ \exists x.\ P a \lor P b \Rightarrow P x)\} \vdash \neg(\forall a b.\ \exists x.\ P a \lor P b \Rightarrow P x)$$

2.2 Conversions

In this phase we convert the conclusion to Conjunctive Normal Form (CNF), and for this we build on a standard set of HOL conversions, originally written by John Harrison in HOL-Light [5] and ported to HOL98 by Donald Syme. In order, the conversions we perform are:

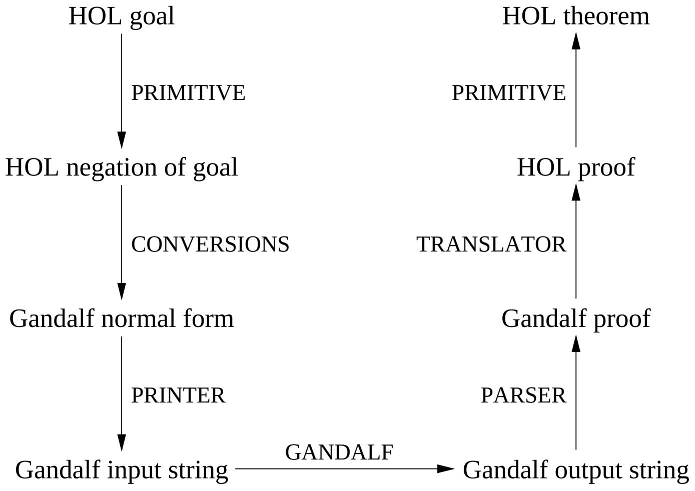


Fig. 3. Overview of GANDALF_TAC.

1. Negation normal form.
2. Anti-prenex normal form.
3. Move existential quantifers to the outside (partial Skolemization).
4. Conjunctive normal form.

After these conversions we extract the conjuncts to give to Gandalf.

Our example becomes:

$$f : (8ab\colon 9x\colon Pa \_ Pb \;)\; Px)g \;‘\; 9ab\colon {}^{\square}8x^{0}\colon :(Px^{0}) \;\hat{}\; {}^{\square}Pa \_ Pb$$

and the relevant terms we will pass to Gandalf are:

$$:(Px^{0})$$
$$Pa \_ Pb$$

2.3 Printing

We now have our goal in a form acceptable to Gandalf, and we can write it to a string and send it to the program. We must add a header and a footer of control information, and we must also take care to rename all the HOL constants/variables, so that constants and existentially quantified variables have names of the form c*n*, and universally quantified variables have names of the form x*n*. Another wrinkle is that Gandalf will not accept propositional variables directly, so we shoehorn them in by pretending that they're 1 place predicates on a constant symbol not mentioned anywhere else.

This is the input string that we send to Gandalf in our example:

```
%-----------------------%
% hol -> gandalf formula %
%-----------------------%
set(auto).
assign(max_seconds,300).
assign(print_level,30).

list(sos).

-c10(x1).

c10(c5) |
c10(c6).
end_of_list.
```

2.4 Calling Gandalf

We are now ready to call Gandalf with the input string, and here we use the Prosper plug-in interface library.

By default, when GANDALF_TAC is loaded a Gandalf internet server is automatically started on the local machine and registered with the system. Communicating with it is simply a matter of calling routines in the plug-in interface; in particular we can spawn a new Gandalf process, send it an input string, and receive its output as a string.

However, the generic system is as detailed in Fig. 2. The Gandalf internet server can be run on any machine on the network as long as its location is registered with the plug-in interface when the client initialises. In this general setup we can have HOL sessions on several machines all making use of the same Gandalf server. The default is for ease of installation only.

Here is what the Gandalf server returns in our example:

```
Gandalf v. c-1.0c starting to prove: gandalf.26884

strategies selected:
((binary 30 #t) (binary-unit 90 #f) (hyper 30 #f)
(binary-order 15 #f) (binary-nameorder 60 #f 1 3)
(binary-nameorder 75 #f))

********* EMPTY CLAUSE DERIVED *********

timer checkpoints: c(2,0,28,2,30,28)

1 [] c10(c5) | c10(c6).
2 [] -c10(x).
3 [binary,1,2,binary_s,2] contradiction.
```

Note that the variable `x1` we passed in to Gandalf has disappeared, and a new variable `x` has appeared. In general we can assume nothing about the names of variables before and after the call.

2.5 Parsing

Our task is now to parse the output string into a Gandalf proof structure, ready to be translated into the corresponding HOL proof. We first check that the string \EMPTY CLAUSE DERIVED" occurs in the output string (or else the tactic fails), and then cut out the proof part of the output string. The use of ML parser combinators enabled a parser to be quickly constructed, and we put the result into special purpose datatypes for storing proof steps, simplifications and Gandalf terms.

Here is the result of the parse on our running example:

```
[(1, (Axiom(), []),
   [(true, Branch(Leaf "c10", Leaf "c5")),
    (true, Branch(Leaf "c10", Leaf "c6"))]),
 (2, (Axiom(), []),
   [(false, Branch(Leaf "c10", Leaf "x"))]),
 (3, (Binary((1, 1), (2, 1)), [Binary_s(2, 1)]),
   [])]
: (int * (Proofstep * Clausesimp list) * (bool * Tree) list) list
```

2.6 Translating

The proof translator is by far the most complicated part of `GANDALF_TAC`. Gandalf has four basic proof steps (binary resolution, hyper-resolution, factoring and paramodulation) and four basic simplification steps (binary resolution, hyper-resolution, factoring and demodulation). Each Gandalf proof line contains exactly one proof step followed by an arbitrary number of simplification steps to obtain a new clause (which is numbered and can be referred to in later proof lines). The proof is logged in detail and in addition after each proof line the desired clause is printed, allowing a check that the line has been correctly followed.

The problem is that even though the proofs are logged in detail, they are occasionally not logged in enough detail to make them unambiguous. The situations in which they are ambiguous are rare, usually involving large clauses when more than one disjunct might match a particular operation, but they occur often enough to make it necessary to tackle the issue.

To illustrate the problem, there may be several pairs of disjuncts that it is possible to factor, or simplifying with binary resolution may be applicable to more than one disjunct in the clause. Gandalf also freely reorders the disjuncts in the axioms with which it has been supplied, requiring some work to even discover which of our own axioms it is using![3]

[3] In addition, my version of Gandalf had a small bug in the proof logging routine requiring some guesswork to determine the exact literals used in the hyper-resolution

At this point it would have been easy to contact the author of Gandalf and ask him to put enough detail into the proofs to completely disambiguate them. However, we decided to remain faithful to the spirit of the project by treating Gandalf as a black-box, and looked instead for a solution within GANDALF_TAC.

We implement a Prolog-style depth-first search with backtracking to follow each line, if necessary trying all possible choices to match the Gandalf clause with a HOL theorem. If there were many ambiguities combined with long proof lines, this solution would be completely impractical, fortunately however ambiguous situations occur rarely and there are usually not many possible choices, so efficiency is not a key question. The only ambiguity that often needs to be resolved is the matching of axioms and this is performed in ML, but all other proof steps are performed as HOL inferences on theorems.

The final line in the Gandalf proof is a contradiction, so the corresponding line in the HOL world is too:

$$f : (8ab : 9x : Pa \_ Pb \ ) \ Px)g \ ` \ ?$$

2.7 Final Primitive Steps

After translation, we need only use the contradiction axiom in order to establish our original goal:

$$` \ 8ab : 9x : Pa \_ Pb \ ) \ Px$$

3 Results

3.1 Performance

In Table 1 we compare the performance of GANDALF_TAC with MESON_TAC, using a set of test theorems taken mostly from the set that John Harrison used to test MESON_TAC, most of which in turn are taken from the TPTP (Thousands of Problems for Theorem Provers) archive[4]. In each line we give the name of the test theorem, followed by the (real) time in seconds to prove the theorem and the number of HOL primitive inference steps performed, for both the tactics. In addition, after the GANDALF_TAC primitive inferences, we include in brackets the number of these inferences that were wasted due to backtracking. As can be seen the number of wasted inferences is generally zero, but occasionally an ambiguity turns up that requires some backtracking.

The other thing to note from Table 1 is that MESON_TAC beats GANDALF_TAC in almost every case, the only exceptions lying in the hard end of both sections. Why is this? Table 2 examines how the time is spent within both tactics. We divide up the proof time into 3 phases:

steps. Of course this can be easily fixed by the author, but it is a good illustration of the type of problem encountered when using an off-the shelf prover.

[4] The TPTP homepage is http://wwwjessen.informatik.tu-muenchen.de/~tptp/.

Table 1. Performance comparison of `GANDALF_TAC` with `MESON_TAC`.

Goal	`MESON_TAC`		`GANDALF_TAC`		
Non-equality					
T	0.027	31	0.034	32	({)
P _ :*P*	0.075	72	2.003	108	(0)
`MN_bug`	0.122	166	2.062	286	(0)
`JH_test`	0.137	176	2.762	312	(0)
`P50`	0.159	243	1.638	441	(0)
`Agatha`	0.438	872	3.916	1891	(0)
`ERIC`	0.989	490	2.750	1268	(0)
`PRV006_1`	1.064	1501	41.044	8097	(109)
`GRP031_2`	1.267	713	8.083	3699	(251)
`NUM001_1`	2.090	1138	40.190	4019	(0)
`COL001_2`	2.150	847	39.240	2620	(0)
`LOS`	5.705	917	5.110	2565	(0)
`GRP037_3`	7.149	2151	79.370	11988	(0)
`NUM021_1`	7.535	1246	17.210	4352	(0)
`CAT018_1`	12.226	2630	61.585	13477	(0)
`CAT005_1`	63.849	2609	66.200	13371	(0)
Equality					
x = *x*	0.090	54	0.041	35	({)
`P48`	0.394	636	2.707	495	(0)
`PRV006_1`	0.648	1053	13.558	4015	(0)
`NUM001_1`	0.768	876	7.032	3012	(0)
`P52`	1.157	1122	{	{	({)
`P51`	1.294	1079	{	{	({)
`GRP031_2`	1.377	757	7.946	3699	(251)
`GRP037_3`	3.402	1466	26.844	8242	(0)
`CAT018_1`	7.646	1809	28.560	8611	(0)
`NUM021_1`	7.737	1026	10.765	3423	(0)
`CAT005_1`	30.514	1784	29.490	8505	(0)
`COL001_2`	56.948	700	4.930	1273	(0)
`Agatha`	{	{	12.626	3409	(0)

{ Conv: Conversion into the required input format.
{ Proof: Proof-search using native datatypes.
{ Trans: Translation of the proof into HOL.

All the entries in this table are geometric means, so the first line represents the geometric means of times for phases of `GANDALF_TAC` to prove all the non-equality problems in Table 1. Geometric means were chosen here so that ratios are meaningful, and the large difference between `GANDALF_TAC` and `MESON_TAC` is in the translation phase; `GANDALF_TAC` struggles to translate proofs in time comparable to finding them, but for `MESON_TAC` they are completely insignificant.

Another interesting difference is in the proofs of equality formulae; `MESON_TAC` has a sharp peak in this entry, but there is no anologue of this for `GANDALF_TAC`.

Table 2. Breakdown of time spent within GANDALF_TAC and MESON_TAC.

	Conv.	Proof	Trans.	Total
GANDALF_TAC				
Non-equality	1.67	5.55	2.14	11.57
Equality	4.20	5.27	7.19	17.82
Combined	2.51	5.42	3.64	13.99
MESON_TAC				
Non-equality	0.26	0.36	0.06	0.90
Equality	0.43	1.27	0.09	2.61
Combined	0.33	0.66	0.07	1.50

It seems likely that this is because Gandalf's has built-in equality reasoning, whereas equality has to be axiomatised in formulae sent to MESON_TAC.

3.2 Gandalf the Plug-In

Putting aside performance issues for the moment, the project has also contributed to the development of the plug-in concept. GANDALF_TAC provides evidence that plug-ins can coexist with the idea of an LCF logical core, and that efficient proving does not have to mean accepting theorems on faith from an oracle.

GANDALF_TAC has also tested the plug-in interface code, which is simple to use, enabling one to concentrate on proof issues without having to think about system details. Once the interface becomes part of the standard HOL distribution then any Gandalf user will be able to download the GANDALF_TAC ML source and wrapper shell script, and use Gandalf in their HOL proofs. Hopefully we will see many more plug-ins appearing in the future.

4 Conclusion

The most important thing to draw from this project is the need for good standards. If Gandalf used a good standard for describing proofs that retained the human-readable aspect but was completely unambiguous, then the project would have been easier to complete and the result would be more streamlined. On the positive side, Gandalf has a good input format that is easy to produce, and the Plug-in interface is an example of a good development standard, taking care of system issues and allowing the programmer to think on a more abstract level. If either of these had been absent, then the project would have been stalled considerably. A program can only be useful as a component of a larger system if the interface is easy for machines as well as people, i.e., simple and unambiguous as well as short and readable.

What is the future for GANDALF_TAC? There are several ways in which the performance could be improved, perhaps the most effective of which would be to enter into correspondence with the author of Gandalf, so that completely explicit (perhaps even HOL-style) proofs could be emitted. Another approach would be to reduce the number of primitive inferences performed, both the initial conversion to CNF and proof translation could be improved in this way. Perhaps they could even be performed outside the logic, minimizing the primitive inferences to a fast verification stage once the right path had been found. There is much scope for optimization of GANDALF_TAC on the ML/HOL side, and the results obtained suggest that such an effort might be sufficient for Gandalf's natural advantages of built-in equality reasoning and coding in C to allow GANDALF_TAC to overtake MESON_TAC in some domains (e.g., hard problems involving equality reasoning).

To look at another angle, GANDALF_TAC is a step towards distributed theorem-proving. It is easy to imagine several proof servers (perhaps several Gandalf servers each with different strategies selected), and a client interface designed to take the first proof that returned and throw the rest away. Distributing such a CPU-intensive, one time activity as theorem-proving makes economic sense, although there are many problems to be solved here, such as how to divide up a problem into pieces that can be seperately solved and joined together at the end.

Finally, one thing that was obvious while developing GANDALF_TAC is that a tool like Gandalf does not really fit in with the interactive proof style popular at present. If Gandalf is used in a proof and takes 3 minutes to prove a subgoal, then every time the proof is run there will be a 3 minute wait. What would perhaps be useful is to save proofs, so that only the speedy verification is run in the future, not the extensive search. Maybe then we would begin to see more tools like Gandalf applied in proofs, as well as the fast tactics that currently dominate.

5 Acknowledgements

I would like to thank Konrad Slind for helping me with all aspects of this work; when I began I was but a HOL novice. Mike Gordon, my supervisor, also gave me valuable advice on how to attack the problem and make the most of the opportunity. Michael Norrish made some useful additions to the Prosper plug-in interface to solve some of the problems I was having, and Myra VanInwegen, Marieke Huisman and Daryl Stewart sympathetically listened to my frequent status reports in the tea-room. Larry Paulson kindly postponed my scheduled group talk when it was obvious that I needed another week. Finally, the comments from the anonymous referees greatly improved the final version.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97{116. Kluwer, 1998.

[2] H. Busch. First-order automation for higher-order-logic theorem proving. In *Higher Order Logic Theorem Proving and Its Applications*, volume Lecture Notes in Computer Science volume 859. Springer-Verlag, 1994.

[3] M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

[4] J. Harrison. Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 313{327, New Brunswick, NJ, 1996. Springer-Verlag.

[5] J. Harrison. *The HOL-Light Manual (1.0)*, May 1998. Available from http://www.cl.cam.ac.uk/users/jrh/hol-light/.

[6] R. Kumar, Th. Kropf, and Schneider K. Integrating a first-order automatic prover in the hol environment. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications (HOL91)*, pages 170{176, Davis, California, August 1991. IEEE Computer Society Press.

[7] K. Slind. *HOL98 Draft User's Manual, Athabasca Release, Version 2*, January 1999. Available from http://www.cl.cam.ac.uk/users/kxs/.

[8] T. Tammet. A resolution theorem prover for intuitionistic logic. In *Cade 13*, volume Lecture Notes in Computer Science volume 1104. Springer Verlag, 1996.

[9] T. Tammet. *Gandalf version c-1.0c Reference Manual*, October 1997. Available from http://www.cs.chalmers.se/~tammet/.

[10] T. Tammet. Towards efficient subsumption. In *Automated Deduction: Cade 15*, volume Lecture Notes in Artificial Intelligence volume 1421. Springer, 1998.

Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving

Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger

Strategic CAD Labs, Intel Corporation, JFT-104
5200 NE Elam Young Parkway, Hillsboro, OR 97124

Abstract. Combining theorem proving and model checking offers the tantalizing possibility of efficiently reasoning about large circuits at high levels of abstraction. We have constructed a system that seamlessly integrates symbolic trajectory evaluation based model checking with theorem proving in a higher-order classical logic. The approach is made possible by using the same programming language (fl) as both the meta and object language of theorem proving. This is done by \lifting" fl, essentially deeply embedding fl in itself. The approach is a pragmatic solution that provides an efficient and extensible verification environment. Our approach is generally applicable to any dialect of the ML programming language and any model-checking algorithm that has practical inference rules for combining results.

1 Introduction

Even with the continuing advances in model-checking technology, industrial-scale verification efforts immediately confront model-checking capacity limits. Using theorem proving to compose verification results offers the tantalizing possibility of mitigating the capacity limitations of automated model checking. The goal is to build verification systems that provide both the power of theorem proving to reason about large systems at high levels of abstraction and the efficiency and usability of model checking.

In this paper, we describe a combined model-checking and theorem-proving system that we have built on top of the Voss system [16]. The interface language to Voss is fl, a strongly-typed functional language in the ML family [23]. Model checking in Voss is done via symbolic trajectory evaluation (STE) [26]. Theorem proving is done in the ThmTac[1] proof tool. ThmTac is written in fl and is an LCF-style implementation of a higher-order classical logic. Although much of our discussion includes references to fl and trajectory evaluation, our approach is generally applicable to any dialect of ML and any model-checking algorithm that has practical inference rules for combining results.

Our goal in combining trajectory evaluation and theorem proving was to move seamlessly between model checking, where we *execute* fl functions, and

[1] The name \ThmTac" comes from \theorems" and \tactics".

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 323{340, 1999.

theorem proving, where we *reason* about the behavior of fl functions. At the same time, we wanted users who only want to do model-checking to be unencumbered by artifacts of theorem-proving. These goals are achieved via a mechanism that we refer to as \lifted fl".

The basic concept of lifted fl is to use fl as both the object and meta language of a proof tool. Parsing an fl expression results in a conventional combinator graph for evaluation purposes and an abstract syntax tree representing the text of the expression. The abstract syntax tree is available for fl functions to examine, manipulate, and evaluate. Lifted fl is similar to reflection, or the quote mechanism in Lisp. One important difference from Lisp is that fl is strongly typed, while Lisp is weakly typed. Our link from theorem proving to model checking is via the evaluation of lifted fl expressions. Roughly speaking, any fl expression that evaluates to true can be turned into a theorem.

Figure 1 shows how we move smoothly between standard evaluation (i.e., programming) and theorem proving. The left column illustrates the proof of a trivial theorem. We first evaluate `1 + 4 - 2 > 2` to convince ourselves that it is indeed true. We then use `Eval_tac` to prove this theorem in a single step. The right column illustrates how we debug a proof that fails in the evaluation step by executing fl code from the proof. Also note that ThmTac does not affect the use of trajectory evaluation. In fact, someone who does not want to do theorem proving is completely unaware of the existence of ThmTac.

```
fl> 1 + 4 - 2 > 2;                    fl> Prove '1 + 4 - 2 > 12' Eval_tac;
it :: bool                            it :: Theorem
T                                     *** Failure:    Eval_tac

fl> Prove '1 + 4 - 2 > 2' Eval_tac;  fl> 1 + 4 - 2;
it :: Theorem                         it :: int
|- '1 + 4 - 2 > 2'                    3
```

Transition from evaluation
to theorem proving

Debug proofs by evaluation

Fig. 1. Example use of evaluation and lifted-fl

Before proceeding, we must introduce a bit of terminology. Trajectory evaluation correctness statements are of the form STE *ckt* A C. The *antecedent* (A) gives an initial state and input stimuli to the circuit *ckt*, while the *consequent* (C) specifies the desired response of the circuit. Trajectory evaluation is described in Section 2 and detailed in other papers [15, 16, 26].

Figure 2 illustrates the transition between model checking and theorem proving. In the left column, we check if a trajectory evaluation run succeeds and then convert it into a theorem. In the right column, we debug a broken proof by executing the trajectory evaluation run that fails. When a trajectory evaluation run does not succeed, it returns a BDD describing the failing condition. In this example, we see that the circuit does not match the specification if `reset` is true or if `src1[63]` and `src2[63]` are both false.

```
fl> STE ckt A C;
it :: bool
running STE....success!
T

fl> Prove ‘STE ckt A C’ Eval_tac;
it :: Theorem
running STE....success!
|- STE ckt A C
```

Transition from model checking to theorem proving

```
fl> Prove ‘STE ckt A D’ Eval_tac;
it :: Theorem
running STE....DID NOT SUCCEED
*** Failure:     Eval_tac

fl> STE ckt A C;
it :: bool
running STE....DID NOT SUCCEED
reset + !src1[63]&!src2[63]
```

Examine STE counter example if `Eval_tac` fails

Fig. 2. Example use of model checking and theorem proving

As a motivating example for the use of theorem proving, consider the inference rule for combining two consequences together, STE_and (Figure 3). This rule says: if we can use the antecedent A to prove consequent C_1 and, in a separate verification run, use A to prove C_2; then we can use A to prove the conjunction of C_1 and C_2. This rule can substantially reduce the complexity of a trajectory evaluation run because, unlike CTL model checking, the complexity is dependent on the size of the *specification* and not the size of the *circuit*. Additional trajectory evaluation rules include pre-condition strengthening, post-condition weakening, transitivity, and case-splitting [2, 15]. After using the inference rules to decompose a proof obligation into a set of smaller trajectory evaluation goals, we use `Eval_tac` to carry out the individual trajectory evaluation runs.

$$\frac{\mathsf{STE}\ ckt\ A\ C_1; \qquad \mathsf{STE}\ ckt\ A\ C_2}{\mathsf{STE}\ ckt\ A\ (C_1\ \mathsf{and}\ C_2)}\ \mathsf{STE\_and}$$

Fig. 3. STE conjunction inference rule

We have used the combination of trajectory evaluation and theorem proving in several large verification efforts. Published accounts include a variety of floating-point circuits and an IA-32 instruction-length decoder [1, 2, 22]. In contrast to previous papers that focused on the use of theorem proving to manage model checking, this paper describes the implementation aspects of the system.

Our proof tool, ThmTac, roughly follows the model of LCF proof systems [13]. The core of ThmTac is encapsulated in an abstract datatype and all extensions to the core are definitional, which provides both flexibility and confidence in the soundness of the system. The core of ThmTac is a set of trusted tactics and is not

not fully expansive, in that proofs do not expand to trees of primitive inference rules. Tactics work backwards and do not provide a forwards-proof justification of their action.

1.1 Related Work

A number of groups have attempted to combine theorem proving and model checking; most have found it to be a surprisingly problematic undertaking. There have been several attempts to use a model checker as a decision procedure in a proof system [6, 10, 14, 24]. Those efforts have not targeted large-scale gate-level hardware verification. As described in Section 2.3, experiences with HOL-Voss [19] demonstrated that such an approach does not provide an effective platform for gate-level hardware verification.

Automated model checkers sometimes include reduction rules that transform the model and/or specification to decrease the size of the state space. These rules are typically hard-coded into the implementation, not available for interactive use, and not integrated with general purpose reasoning facilities [8, 18, 21].

ACL2 [20], originally from Computational Logic, is the work most closely related to that presented here, in that Applicative Common Lisp is used as the implementation language and the term language. ACL2 has been used for large-scale theorem-proving based hardware verification [25], but this work has not made use of model-checking.

2 Evolution of a Solution

In this section we describe the lessons learned during the evolution of symbolic trajectory evaluation over the course of almost ten years. During that time, two different combinations of trajectory evaluation and theorem proving were implemented. The next section describes how lifted-fl and the third experiment, ThmTac, resulted from the lessons learned.

2.1 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation [26] is based on traditional notions of digital circuit simulation and excels at datapath verification. Computing $\mathsf{STE}\ \mathit{ckt}\ A\ C$ gives the weakest condition under which the antecedent A satisfies the consequent C. One of the keys to the efficiency of trajectory evaluation and its success with datapath circuits is the restricted temporal logic. The core specification language for antecedents and consequences (trajectory formulas) is shown in Figure 4.

For the purposes of this paper, the most important aspect of trajectory evaluation is the existence of a set of inference rules for manipulating trajectory formulas. These inference rules allow us to combine results together [16], and to use the *parametric representation* (Section 4.3) to transform trajectory formulas to increase model checking efficiency [2].

```
traj_form   node is value
          j traj_form when guard
          j N traj_form
          j traj_form and traj_form
```

The meaning of the trajectory formula: N^t(*node* is *value* when *guard*) is: \if *guard* is true then at time *t*, *node* has value *value*"; where *node* is a signal in the circuit and *value* and *guard* are Boolean expressions (BDDs).

Fig. 4. Mathematical definition of trajectory formulas

2.2 Voss

Symbolic trajectory evaluation was first implemented as an extension to the COSMOS symbolic simulator [5]. After several experiments with different forms of interaction with the simulator, Seger created the Voss implementation of trajectory evaluation and included an ML-like language called \fl" as the interface. In fl, the Boolean datatype is implemented using Binary Decision Diagrams (BDDs) [7]. Trajectory evaluation is executed by calling the built-in function `STE`.

Voss provides a very tight and efficient verification and debug loop. Users can build scripts that run the model-checker to verify a property, generate a counter example, simulate the counter-example, and analyze the source of the problem. Experience has shown that Voss provides a very efficient platform for debugging circuits.

Trajectory formulas are implemented as a very shallow embedding in fl, rather than as a more conventional deep embedding. Trajectory formulas are implemented as lists of *five-tuples* (Figure 5). This style of embedding has advantages both for model checking extensibility and for theorem proving. For example, users generally find the definitions of `from` and `to` preferable to nested \nexts".

```
==                          guard    node     value    start    end
lettype traj_form = (  bool     string    bool     nat      nat ) list

let n is v      = [(T; n; v; 0; 1)]
let f1 and f2   = f1 append f2
let f  when g   = map (  (g'; n; v; t0; t1): (g' AND g; n; v; t0; t1)) f
let N f         = map (  (g; n; v; t0; t1): (g; n; v; t0 + 1; t1 + 1)) f

let f from t0   = map (  (g; n; v; z; t1): (g; n; v; t0; t1)) f
let f to t1     = map (  (g; n; v; t0; z): (g; n; v; t0; t1)) f
```

Fig. 5. Implementation of trajectory formulas in Voss

The value field in a trajectory formula can be any Boolean proposition (BDD). Because of the shallow embedding, users have the complete freedom of a general-purpose functional programming language in which to write their specifications. As users of proof systems have known for a long time, this facilitates concise and readable specifications.

The orthogonality of the temporal aspect (the two time fields) of the specification from the data computation (the guard and value fields) has a number of positive ramifications. Although trajectory formulas are not generally executable, the individual fields are executable. For example, users can evaluate the data aspect of their specification simply by evaluating the fl function that computes the intended result. Standard rewrite rules and decision procedures over Booleans can be applied to the guard and data fields. Rewrite rules and decision procedures for integers can be applied to the temporal fields. A standard deep embedding of a temporal logic would require specialized rewrite rules and decision procedures for trajectory formulas.

In summary, the lessons learned as Voss evolved from simply an implementation of trajectory evaluation to a general-purpose platform for symbolic simulation and hardware verification were:

1. An ML-like programming language (such as fl) makes an excellent interface to a verification system.
2. The shallow embedding of the specification language in a general-purpose programming language facilitates user extensibility and concise, readable specifications.
3. An executable specification language makes it much easier to test and debug specifications.

2.3 HOL-Voss

In 1991, Joyce and Seger constructed the HOL-Voss system [19], which was an experiment with using trajectory evaluation as a decision procedure for the HOL proof system [13]. When the model checker is just a decision procedure in the proof system, the user's only method of interacting with it is to call the model checker as a tactic. This isolates the user from the powerful debugging and analysis capabilities built into the model checker. In addition, practical model checking is rarely, if ever, a black box procedure that comes back with \true" or \false". Rather, model checking involves significant manual interaction, dealing with a variety of issues including BDD variable orders and environment modeling. Consequently, using a model-checker as a decision procedure in a proof system does not result in an effective hardware verification environment.

The key lessons learned from HOL-Voss were:

1. From the same prompt, the user must be able to interact directly with either the model checker or the proof tool.
2. More time is spent in model checking and debugging than in theorem proving.

2.4 VossProver

Applying lessons learned from HOL-Voss and new inference rules for trajectory evaluation, Seger and Hazelhurst created VossProver, a lightweight proof system built on top of Voss [15].

VossProver was implemented in fl in typical LCF style with an abstract datatype for theorems. Hazelhurst and Seger used a limited specification language to simplify theorem proving. The specification language of VossProver was a deep embedding in fl of Booleans and integers and a shallow embedding of tuples, lists, and other features. The transition from theorem proving to model checking was done by translating the deeply embedded Boolean and integer expressions into their fl counterparts and then evaluating the resulting fl expressions. The success of the approach implemented in VossProver was demonstrated by a number of case studies, including the verification of a pipelined IEEE-compliant floating-point multiplier by Aagaard and Seger [3].

The translation from the deeply-embedded specification language used in theorem proving to the normal fl used in model checking was awkward. Definitions were often duplicated for the two types. The difficulty of evaluating Boolean expressions at the fl prompt was a serious detraction when compared to the ease of use provided by specifications in fl.

The lack of support for types other than Boolean and integers and the inability to define and reason about new identifiers limited the usefulness of the VossProver outside of the range of arithmetic circuits for which it was originally developed. Our conclusions from analyzing the VossProver work were:

1. When model checking, the user must be unencumbered by artifacts from the theorem prover.
2. The specification language for model checking must support simple and useful inference rules.
3. The theorem prover and model checker must use the same specification language.
4. Even if a proof system has a very narrow focus it should include a general-purpose specification language.
5. fl, being a dialect of ML, is a good language for implementing a proof system.

Analyzing the conclusions left us with the seemingly contradictory conclusion that we *wanted to use* fl *as both the specification and implementation language for a proof tool*. Our solution to the contradiction was to deeply embed fl within itself, that is, to \lift" the language. This allows fl programs to see and manipulate fl expressions. The remainder of the paper describes lifted fl and its use in implementing ThmTac, the current proof system built on top of Voss.

3 Implementation of Lifted-fl

A programming language sees the *value* of an expression and not the structure of the expression. For example, the value of 3 + 4 is 7. By examining the value, we cannot distinguish 3 + 4 from 8 − 1. An equality test in a programming language is a test of *semantic equality*. In a programming language, every time we evaluate an *if-then-else*, we \prove" that the condition is semantically equal to true or false.

In a logic, the primitive inference rule for equality of two terms is a test of *syntactic* equality (e.g., a = a). The equality symbol in the logic forms an assertion about *semantic* equality (e.g., a + b = b + a). The power of theorem proving comes from the ability to use syntactic manipulation to prove the semantic equality of expressions that cannot be evaluated. The inspiration behind lifted-fl was to allow this type of reasoning about fl programs themselves.

By basing ThmTac on lifted fl, we created a world where we can (1) directly and efficiently evaluate terms in the specification language and (2) manipulate and reason about them. This effectively merges the worlds of semantic evaluation in model checking and syntactic manipulation in theorem proving.

We lift an fl expression by enclosing it in backquotes (e.g., ‘3+4‘). If an expression has type , the lifted expression has type expr (e.g., ‘3 + 4‘ :: int expr). We evaluate concrete lifted-fl expressions with the full efficiency of normal fl code via the function eval :: expr ! , which takes an expression and returns a value of type .

The fl programming language is a strongly-typed, lazy functional programming language. Syntactically, it borrows heavily from Edinburgh-ML [13] and semantically, its core is very similar to lazy-ML [4]. One distinguishing feature of fl is that Binary Decision Diagrams (BDDs) [7] are built into the language and every object of type bool is represented as a BDD.[2]

Similar to most functional languages, the input language (fl) is first translated to a much simpler intermediate language that is input to the compiler and interpreter. For example, during this first stage, functions defined by pattern matching are translated into functions that perform the pattern matching explicitly using standard pattern matching compilation [12]. The intermediate language is essentially an enriched form of lambda expressions.

Figure 6 shows the data type representing the intermediate language for fl. All but two of the constructors are similar to that of a generic higher-order logic proof system [13]. The expression constructor (EXPR) is used to represent lifted terms. A user-defined function definition (USERDEF) contains the name of the identifier (the string), the body of the definition (the term), and the fl graph to execute (the code).

To illustrate, Figure 7 gives an overview of the implementation of lifted fl.[3] The top of the figure shows four lines of fl code: the definition of the function inc, a call to inc with an argument of 14, a lifted version of the same expression with

[2] Strictly speaking, the type of these objects should be env ! bool, where env is an interpretation of the variables used in the BDD. However, for convenience the global environment is kept implicit and the type abbreviated to bool.

[3] For conciseness, Figure 7 does not include all of the fields for USERDEF and glosses over the fact that leaf terms (e.g., INT 14) should be encapsulated by a LEAF constructor (e.g., (LEAF (INT 14))).

```
lettype leaf =                          andlettype term =
     INT int                               APPLY term term
   | STRING string                       | LAMBDA string term
   | BOOL bool                           | LEAF leaf
   | PRIM_FN int                         | EXPR term
   | USERDEF string term code          ;
   | VAR string
   | NIL
```

Fig. 6. Term type in lifted fl

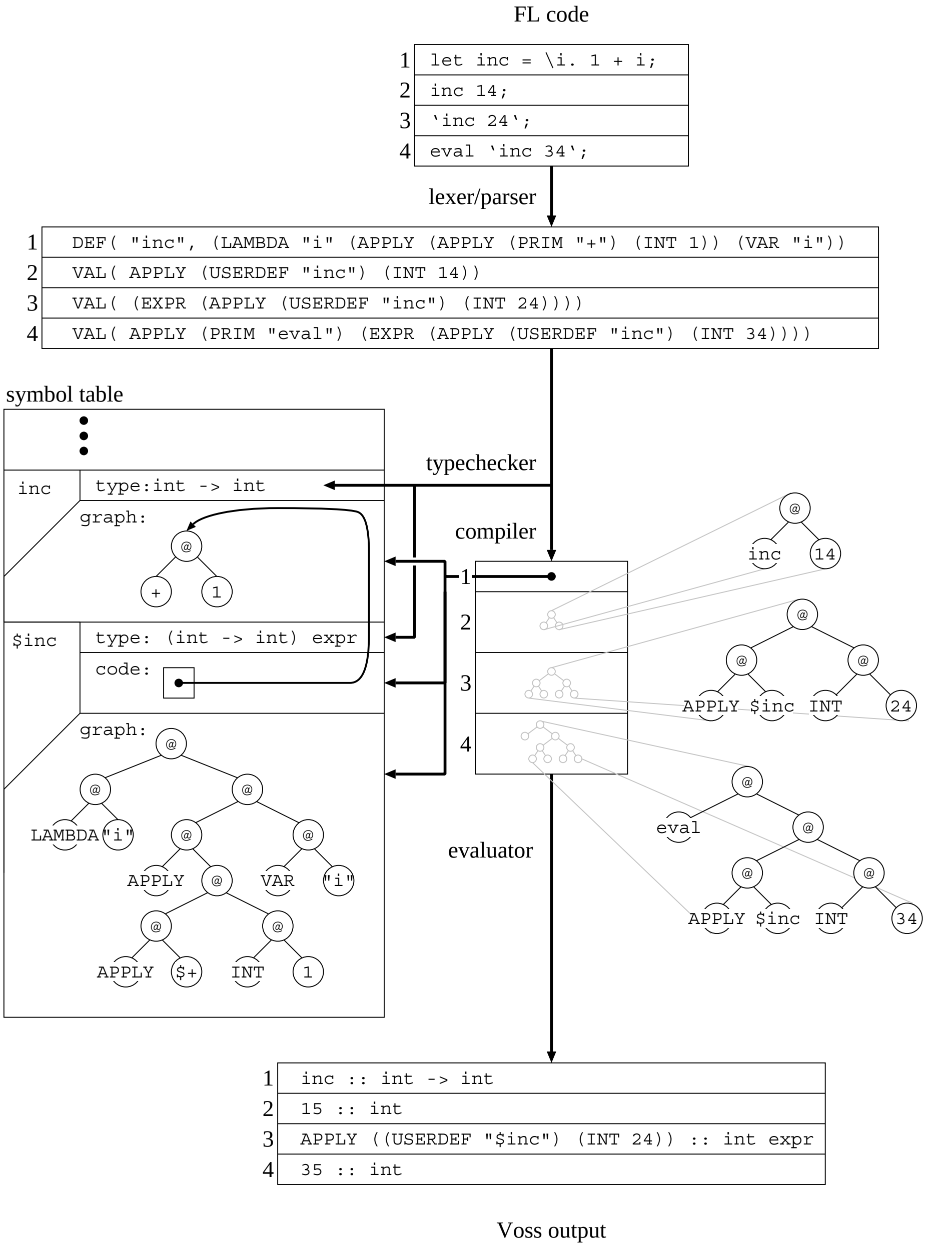
an argument of 24, and the evaluation of the lifted expression with an argument of 34. The Voss output for each of the four lines is shown at the bottom of the figure. Between the input and the output are the intermediate steps and the symbol table.

The parser generates abstract syntax trees that are identical to those of a conventional parser with the exception of EXPR, which is used to mark that an expression is to be lifted. For each fl identifier, the typechecker adds two entries to the symbol table, one for the regular identifier (inc) and one for the lifted identifier ($inc). The entry for the regular identifier contains two fields: *type* and *graph*. The *type* field is inserted by the typechecker, and the *graph* field is the combinator graph generated by the compiler. The lifted identifier contains three fields: *type*, *graph*, and *code*. The *type* field is the same as for the regular function except that it is made into an expression type. The *graph* field for the lifted identifier is the concrete data-type representation of the intermediate language (Figure 6). The *code* field is included for efficiency reasons, and is a pointer to the *graph* field of the regular identifier.

The abstract syntax trees are semantically identical to the data structure of lifted fl. For example, in Figure 7, compare the second line generated by the parser and the third line generated by the evaluator. The abstract syntax tree of the non-lifted expression matches the resulting value of the lifted expression.

To evaluate a lifted-fl expression, we first replace the lifted combinator graph nodes with their non-lifted counterparts. The code field of USERDEF improves the efficiency of evaluation. We use the code field to replace occurrences of lifted identifiers (e.g., $inc) with the graph of the non-lifted identifier, rather than translate the graph of the lifted identifier to its non-lifted counterpart. After translating the graph, we call the fl evaluator as with conventional evaluation.

So far, we have described lifted fl as the exact counterpart of regular fl. To support theorem proving, lifted fl allows free variables | which are not allowed in regular fl. Free variables are identifiers prefixed by \?" or \!" (e.g., ?x and !f). We modified the typechecker so that it knows whether it is in lifted or regular mode. The two modes are essentially the same, except that the regular mode complains when it encounters a free variable while the lifted mode does not complain if the first character of the free variable is \?" or \!". As expected,



FL code

```
1 let inc = \i. 1 + i;
2 inc 14;
3 `inc 24`;
4 eval `inc 34`;
```

```
1 DEF( "inc", (LAMBDA "i" (APPLY (APPLY (PRIM "+") (INT 1)) (VAR "i"))
2 VAL( APPLY (USERDEF "inc") (INT 14))
3 VAL( (EXPR (APPLY (USERDEF "inc") (INT 24))))
4 VAL( APPLY (PRIM "eval") (EXPR (APPLY (USERDEF "inc") (INT 34))))
```

```
1 inc :: int -> int
2 15 :: int
3 APPLY ((USERDEF "$inc") (INT 24)) :: int expr
4 35 :: int
```

Voss output

Fig. 7. Implementations of lifted fl

evaluating a lifted-fl term raises an exception if the associated expression contains any free variables.

4 ThmTac

The logic of ThmTac is a higher-order classical logic. The term language is the lambda calculus extended with the primitive functions of fl. Much of the implementation of ThmTac is similar to that of a typical LCF system. We assume that the reader is familiar with the implementation of a typical LCF proof system. We will focus our discussion on those aspects of ThmTac that differ from convention.

4.1 Basic Term Language

As described at the end of Section 3, we modified fl to support free variables in lifted fl. Quantifiers are another distinction between a programming language and a logic. We implemented quantifiers in ThmTac with functions that raise exceptions when evaluated and then axiomatized their behavior.

One of the few basic term operations underlying ThmTac that differs from a conventional implementation is that of \unfolding" a definition. Recall that in the `USERDEF` constructor, identifiers carry their body with them. Rather than manipulating symbol tables and concerning ourselves with scoping rules and multiple definitions, we simply replace the occurrence of the identifier with the body from the `USERDEF`.

4.2 General Tactics

The core set of tactics, tacticals, rewrites and conversionals are trusted and enclosed in abstract datatypes. Inside the core, ThmTac tactics are functions from a sequent to a list of sequents. An empty sequent list means that the tactic solved the goal. Outside of the core, users combine trusted pieces together in the conventional LCF style. This hybrid approach strikes a good balance between soundness and productivity (both of system development and verification).

We use three abstract datatypes to protect the core: `Theorem`, `Tactic`, and `Rewrite`. Slightly simplified versions of these types and several accessor functions are shown in Figure 8.

In the LCF style, we provide users the full power of a programming language (fl) to construct new tactics and tacticals from existing tactics and tacticals. Because tactics are encapsulated in an abstract datatype, they cannot be be directly applied to sequents. We provide the function `apply_tac` to apply a tactic to a sequent. The function `peek_at_sequent` allows users to examine the sequent when choosing which tactics to apply without compromising the safety of the system.

```
lettype Sequent = SEQUENT (bool expr list) (bool expr);
lettype Theorem = THEOREM (bool expr);
lettype Tactic  = TACTIC (sequent ! sequent list);
lettype Rewrite = REWRITE (  expr !    expr);

apply_tac        :: Tactic ! Sequent ! Sequent list
peek_at_sequent :: (Sequent ! Tactic) ! Tactic
Prove            :: bool expr ! Tactic ! Theorem
```

Fig. 8. Implementations of core abstract datatypes

4.3 Special Tactics

Evaluation as a Tactic As described earlier, eval is an fl function of type expr ! . Evaluation is available to the ThmTac user in rewriting and in tactic application. Eval_rw is a rewrite that evaluates a term and substitutes the result in for the original term. Eval_tac evaluates the goal of the sequent and solves the goal if it evaluates to true.

Our primary intention with lifted-fl was to provide for the evaluation of trajectory evaluation runs (e.g., Figure 2). In addition, evaluation can be applied to general expressions, and is easier and more efficient than applying libraries of rewrite rules for each of the different functions to be evaluated.

Because Booleans are built into fl as BDDs, Eval_tac provides an efficient decision procedure for Booleans. Additionally, we have written tactics that take goals with non-Boolean subterms, replace each term containing non-Boolean subterms with a unique Boolean variable, and evaluate the goal.

STE-Specific Tactics There are several trajectory evaluation inference rules for combining and decomposing STE calls [15]. Figure 3 in Section 1 showed the inference rule for combining two trajectory evaluation runs with identical antecedents and different consequents. This and the other trajectory evaluation inference rules are implemented as core tactics in ThmTac.

Parameterization Tactics In an earlier paper [2], we presented an algorithm for computing the *parametric representation* of a Boolean predicate and described its use in conjunction with case splitting in a symbolic simulation environment. The parametric representation differs from the more common characteristic-function representation in that it represents a set of assignments to a vector of Boolean variables as a vector of fresh (parametric) variables, where all assignments to the parametric variables produce elements in the set. For example, a one-hot[4] encoding of a two-bit vector has a parametric representation of $ha_0;\overline{a_0}i$ and a one-hot three-bit vector has a parametric representation of $ha_0;\overline{a_0}a_1;\overline{a_0}\overline{a_1}i$.

[4] A *one-hot* encoding requires that exactly one bit of a k-bit vector is asserted. For a two-bit vector, it is simply XOR.

Figure 9 shows how the three core tactics (`CaseSplit_tac`, `ParamSTE_tac`, `Eval_tac`) are used to integrate the case splits, the parametric representation, and trajectory evaluation.

```
1 ⊢ $P$ ⟹ STE $ckt$ $ant$ $cons$
DO CaseSplit_tac [$P_0$; $P_1$; $P_2$]
    1.1 ⊢ ($P_0$ OR $P_1$ OR $P_2$) ⟹ $P$
     DO Eval_tac
      ■
    1.2 ⊢ $P_0$ ⟹ STE $ckt$ $ant$ $cons$
     DO ParamSTE_tac
        1.2.1 ⊢ STE $ckt$ (param_traj $P_0$ $ant$) (param_traj $P_0$ $cons$)
          DO Eval_tac
           ■
    1.3 ⊢ $P_1$ ⟹ STE $ckt$ $ant$ $cons$
     DO ParamSTE_tac
        1.3.1 ⊢ STE $ckt$ (param_traj $P_1$ $ant$) (param_traj $P_1$ $cons$)
          DO Eval_tac
           ■
    1.4 ⊢ $P_2$ ⟹ STE $ckt$ $ant$ $cons$
     DO ParamSTE_tac
        1.4.1 ⊢ STE $ckt$ (param_traj $P_2$ $ant$) (param_traj $P_2$ $cons$)
          DO Eval_tac
           ■
```

Fig. 9. Example use of case splitting, parametric, and evaluation using core tactics

For an n-way case split, `CaseSplit_tac` generates $n + 1$ subgoals (1.1–1.4). The first subgoal (1.1) is a side condition to show that the case splits are sufficient to prove the original goal. Finally, there is one subgoal for each case (1.2–1.4). By itself, this tactic is similar to other proof systems. However, with the availability of `Eval_tac`, we can usually dispose of the first subgoal automatically and efficiently. `ParamSTE_tac` is implemented via rewriting as expected. There is also a user-level tactic that combines these three tactics together so that the proof shown in Figure 9 can be carried out in a single step.

`Eval_tac` has already been described, but this is a good opportunity to mention an important aspect of its use that has a dramatic impact on the user's view of the system. Almost all proofs and model checking runs fail when first tried. When `Eval_tac` fails, it generates a counter example that the user can analyze. Because theorem proving, model checking, and debugging are conducted in the same environment, the user's infrastructure is available for all three activities. If the counter example comes from a trajectory evaluation run, the user can debug the counter example using the powerful debugging aids for trajectory evaluation available in Voss.

Tactics for Instantiating Terms and BDD Variables One of the critical steps in bridging the semantic gap between theorem proving and model checking is to move from a world of term quantifiers and variables to a world of BDD quantifiers and variables. For example, we could transform: $\forall x. P(x)$ to

`QuantForall "x" (`P`(variable "x"))`

where `QuantForall` is a function that universally quantifies the BDD variable `"x"` in the BDD resulting from evaluating P(`variable "x"`). Consider the goal $\forall x. P(x)$, where x is a term variable of type `bool`. Applying `Eval_tac` will fail, because the function $\forall$ cannot, in general, be evaluated (e.g., consider quantification over integers).

Replacing a term quantifier with a BDD quantifier and BDD variable is not as simple as it seems at first glance, as there are performance and soundness issues. If ThmTac were to provide a new and unique name for the BDD variable, it would not be placed in an optimal location in the all-important BDD-variable order defined by the user. Additionally, increasing the number of BDD variables globally active in the system slows down some BDD operations. Thus, for performance reasons, the user needs to provide the variable name.

If the user provides a variable name that is already used within the proof, then the proof will not be sound. Thus, for correctness purposes, ThmTac has the burden of making sure that the name provided by the user is truly a fresh variable.

The process of replacing term quantifiers and variables with BDD quantifiers and variables and then evaluating the goal is implemented by the tactic `BddInstEval_tac`. The user provides the name of fresh BDD variable (say `"y"`). The tactic first checks that this variable has not yet been used in the proof. If the variable is fresh, `BddInstEval_tac` replaces the term quantifier $\forall x$ with the BDD quantifier `QuantForall "y"`, instantiates x in $P(x)$ with `variable "y"`, and then applies `Eval_tac`.

5 Analysis of the Soundness of Our Approach

When designing ThmTac, our goal was to create a prototype verification system that explored new mechanisms for combining model checking and theorem proving. We tried to strike a balance between soundness and productivity — both in system building and verification. With respect to soundness, we focused our efforts on preventing users from inadvertently proving false statements, but did not exert undue effort in protecting against adversarial users — someone who intends to prove a false statement.

In analyzing the soundness of our design, we started from two facts. First, the typed lambda calculus is a sound logic. Second, a strongly typed, pure functional programming language is very similar to the typed lambda calculus with the inclusion of non-primitive recursion. From this, we focussed on two questions. First, how do we deal with recursion in ThmTac? Second, what additional features does fl include and how do they affect soundness?

5.1 Recursion

To deal with recursion in a truly sound manner, we would need to ensure that the computation of every value in fl terminates. The definition `letrec x = NOT x;` can be used to prove false without much difficulty. Tools such as Slind's TFL [27] could be used to prove termination of fl functions with reasonable effort. Even without such formal support, non-termination does not pose a significant problem for us. First, because we extensively test specifications by evaluating them in fl before attempting to reason about them, termination problems are caught before they lead to soundness problems in theorem proving. Second, most hardware specifications we have used are not recursive.

5.2 Additional Language Features to Be Examined

Section 4 discussed instantiating term quantifiers and variables with BDDs. We list the topic here simply for completeness.

In fl, prior to lifted-fl and ThmTac, testing for the equality of functions returned either true or false. If the two functions did not have the same graph structure, then false was returned. This behavior created a potential way to prove false. For example, the evaluation of:

$$(\lambda x.\ \lambda y.\ x + y) = (\lambda x.\ \lambda y.\ y + x)$$

would return false, while we could rewrite one side of the equality using commutativity of addition and prove that the same statement was true. We dealt with this problem by changing the implementation of fl so that if it cannot determine for certain that two functions are the same, it raises an exception rather than return false.

There are three features of fl that pose theoretical, but not practical, soundness problems: exceptions, catching exceptions, and references. As a practice, exceptions, catches, and references occur only in relatively low-level code. This type of code is evaluated, or the behavior is axiomatized, but the text of the code is not reasoned about. These features are included in fl for implementation of an efficient verification system. Our requirement then, is that the code that uses these features is axiomatized correctly and implemented so that the effects of the features are not visible outside of the code | that is the code appears functional from the outside.

Coquand showed that a consequence of Girard's Paradox is that ML-style polymorphism is unsound [9]. The fl type system does not support *let polymorphism*, which is the distinguishing feature in ML-style polymorphism that makes it unsound.

6 Summary

6.1 Effectiveness of Our Approach

In this section we briefly revisit the verification efforts previously described [1, 2] and relate them to the use of lifted-fl. The combination of ThmTac and trajectory evaluation has been used successfully on a number of large circuits within Intel. Table 1 summarizes the number of latches in the circuit, the maximum number of BDD nodes encountered during the verification, the number of case splits used, and the compute time for the model checking and theorem proving. All of the runs reported were performed on a 120MHz HP[5] 9000 with 786MB of physical memory.

Table 1. Verification Statistics.

	Latches	BDD nodes	Cases	Verification time
IM	1100	4.2M	28	8 hr
Add/Sub Ckt1	3600	2.0M	333	15 hr
Add/Sub Ckt2	6900	1.5M	478	24 hr
Control Ckt	7000	0.8M	126	16 hr

One of the first large circuits we verified was the IM, an IA-32 instruction length decoder [1]. We believe this verification to be one of the most complex hardware verifications completed to date. We started with a very high-level specification of correctness that described the desired behavior of the circuit over an infinite stream of IA-32 instructions. Using induction and rewriting, we decomposed the top-level correctness statement into a base case and an inductive case. The base case was dispatched with additional rewriting and evaluation. The inductive case contained a trajectory formula that that was too complex to be evaluated in a single trajectory evaluation run. To solve this problem, we case split on the internal state of the circuit.

Up to this point, the proof had used conventional theorem proving techniques. Two steps remained to bridge the gap to model checking and finish the proof. We first replaced the term quantifiers and term variables with BDD quantifiers and variables. We then applied the parametric tactic to encode the case splits into the antecedents and consequences of the trajectory assertions and completed the proof with a call to `Eval_tac` for each case. We found the ability to debug our proof script in the same environment as we debugged our model checking runs to be of tremendous benefit.

We verified two IEEE-compliant, multi-precision, floating-point adder/subtracters from Intel processors [2] (Add/Sub Ckt1 and Add/Sub Ckt2 in Table 1). The arithmetic portion of our specification was based upon a textbook algorithm [11]. Specifications for flags and special cases were based on IEEE Standard 754 [17] and proprietary architectural documentation. Again, the use of lifted-fl was crucial. We were able to *lift the specification itself* and then manipulate it inside ThmTac. After rewriting, term manipulation, and parametric-based input-space decomposition, we derived the trajectory evaluation runs that completed the verification.

[5] All trademarks are property of their respective owners.

6.2 Retrospective Analysis

The first efforts to connect trajectory evaluation and theorem proving began approximately eight years ago. Many lessons were learned as various experimental systems were designed, constructed, used, and analyzed in the intervening time. Three keys to success stand out. First, clean and useful inference rules for model checking. Second, very tight integration between theorem proving and model checking with full access to both tools. Third, the use of a general-purpose specification language. Symbolic trajectory evaluation provides the first of these items. Lifted fl allowed us to use the same language for both the object and meta language of a proof system and model checker, which gave us the second and third capabilities.

Acknowledgments

We are deeply indebted to Clark Barrett, Tom Melham, and Donald Syme for their insights and development work that have contributed greatly to the work presented here. Additionally, John Harrison and Ching-Tsun Chou read a draft of the paper and provided helpful comments.

References

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *DAC*, pages 538{541. ACM/IEEE, July 1998.

[2] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC*, July 1999.

[3] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ICCAD*, pages 7{10. IEEE Comp. Soc. Press, Washington D.C. Nov. 1995.

[4] L. Augustson. A compiler for Lazy-ML. In *ACM Symposium on Lisp and Functional Programming*, pages 218{227, 1984.

[5] D. E. Beatty, R. E. Bryant, and C.-J. H. Seger. Synchronous circuit verification - an illustration. In *Advanced Research in VLSI, Proceedings of the Sixth MIT Conference*, pages 98{112. MIT Press, 1990.

[6] K. Bhargavan, C. A. Gunter, E. L. Gunter, M. Jackson, D. Obradovic, and P. Zave. The village telephone system: A case study in formal software engineering. In M. Newey and J. Grundy, editors, *Theorem Proving in Higher Order Logics*, pages 49{66. Springer Verlag; New York, Sept. 1998.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on CAD*, C-35(8):677{691, Aug. 1986.

[8] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Prog. Lang. and Systems*, 16(5):1512{1542, Sept. 1994.

[9] T. Coquand. An analysis of Girard's paradox. In *LICS*, pages 227{236. IEEE Comp. Soc. Press, Washington D.C. June 1986.

[10] P. Curzon, S. Tahar, and O. A. Mohamed. Verification of the MDG components library in HOL. Technical Report 98-08, Australian Nat'l Univ., Comp. Sci., 1998. pages 31{46 In Supplementary Proceedings of TPHOLS-98.

[11] J. M. Feldman and C. T. Retter. *Computer Architecture*. McGraw-Hill, 1994.

[12] A. Field and P. Harrison. *Functional Programming*. Addison Wesley, 1988.

[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, 1993.

[14] E. L. Gunter. Adding external decision procedures to HOL90 securely. In M. Newey and J. Grundy, editors, *Theorem Proving in Higher Order Logics*, pages 143{152. Springer Verlag; New York, Sept. 1998.

[15] S. Hazelhurst and C.-J. H. Seger. A simple theorem prover based on symbolic trajectory evaluation and BDDs. *IEEE Trans. on CAD*, Apr. 1995.

[16] S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification*, chapter 1, pages 3{78. Springer Verlag; New York, 1997.

[17] IEEE. *IEEE Standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985, 1985.

[18] N. C. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41{75, Aug. 1996.

[19] J. Joyce and C.-J. Seger. Linking BDD based symbolic evaluation to interactive theorem proving. In *DAC*, June 1993.

[20] M. Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. on Soft. Eng.*, 1997.

[21] K. McMillan. Minimalist proof assistants: Interactions of technology and methodology in formal system level verification. In G. C. Gopalakrishnan and P. J. Windley, editors, *Formal Methods in CAD*, page 1. Springer Verlag; New York, Nov. 1998.

[22] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Tech. Jour.*, First Quarter 1999. Online at http://developer.intel.com/technology/itj/.

[23] L. Paulson. *ML for the Working Programmer,*. Cambridge University Press, 1996.

[24] S. Rajan, N. Shankar, and M. Srivas. An integration of model checking automated proof checking. In *CAV*. Springer Verlag; New York, 1996.

[25] J. Sawada and W. A. J. Hunt. Processor verification with precise exeptions and speculative execution. In A. Hu and M. Vardi, editors, *CAV*, number 1427 in LNCS, pages 135{146. Springer Verlag; New York, June 1998.

[26] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147{189, Mar. 1995.

[27] K. Slind. Derivation and use of induction schemes in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, pages 275{291. Springer Verlag; New York, Aug. 1997.

Symbolic Functional Evaluation

Nancy A. Day[1] and Jeffrey J. Joyce[2]

[1] Oregon Graduate Institute, Portland, OR, USA
nday@cse.ogi.edu
[2] Intrepid Critical Software Inc., Vancouver, BC, Canada
joyce@intrepid-cs.com

Abstract. Symbolic functional evaluation (SFE) is the extension of an algorithm for executing functional programs to evaluate expressions in higher-order logic. SFE carries out the logical transformations of expanding definitions, beta-reduction, and simplification of built-in constants in the presence of quantifiers and uninterpreted constants. We illustrate the use of symbolic functional evaluation as a "universal translator" for linking notations embedded in higher-order logic directly with automated analysis without using a theorem prover. SFE includes general criteria for when to stop evaluation of arguments to uninterpreted functions based on the type of analysis to be performed. SFE allows both a novice user and a theorem-proving expert to work on exactly the same specification. SFE could also be implemented in a theorem prover such as HOL as a powerful evaluation tactic for large expressions.

1 Introduction

Symbolic functional evaluation (SFE) is the extension of an algorithm for executing functional programs to evaluate expressions in higher-order logic. We use SFE to bridge the gap between high-level, expressive requirements notations and automated analysis by directly evaluating the semantics of a notation in higher-order logic. SFE produces the meaning of a specification in a form that can be subjected to behavioural analysis. In our approach, writing the semantics of a notation is the only step necessary to have access to a range of automated analysis procedures for specifications written in that notation. Bridging this gap allows a novice user to perform some types of analysis even if all queries of the specification cannot be checked automatically.

SFE carries out the logical transformations of expanding definitions, beta-reduction, and simplification of built-in constants in the presence of uninterpreted constants and quantifiers. While these logical transformations can be performed by rewriting in a theorem prover, they do not require the full generality of rewriting. To use an algorithm for evaluating functional programs, two special considerations are needed. First, we handle uninterpreted constants. Special treatment of uninterpreted constants allows us to carry out substitution, a core part of the evaluation algorithm, more efficiently. Quantifiers, such as "forall", are treated for the most part as uninterpreted constants. Second,

Y. Bertot et al. (Eds.): TPHOLs'99, LNCS 1690, pp. 341–358, 1999.

we have defined several distinct and intuitive levels of evaluation that serve as \stopping points" for the SFE algorithm when considering how much to evaluate the arguments of uninterpreted functions. The choice of level provides the user with a degree of control over how much evaluation should be done and can be keyed to the type of automated analysis to be performed.

The following simple example illustrates the idea of symbolic functional evaluation. We use a syntactic variant of the HOL [17] formulation of higher-order logic called S [27].[1] In S, the assignment symbol `:=` is used to indicate that the function on the left-hand side is being defined in terms of the expression on the right-hand side. The function `exp` calculates x^y:

```
exp x y := if (y = 0) then 1 else (x * exp x (y - 1));
```

If we use SFE to evaluate this function with the value of `x` being 2 and the value of `y` being 3, the result is 8. In order to test the behaviour of `exp` for more possible inputs, we make the input `x` symbolic. The constant `a` is an uninterpreted constant. Evaluating `exp a 3`, SFE produces:

```
a * (a * (a * 1))
```

If both of the inputs to `exp` are uninterpreted constants, say `a`, and `b`, then the evaluation will never terminate. SFE can tell the evaluation may not terminate as soon as `exp` is expanded the first time because the argument to the conditional is symbolic, i.e., it cannot determine whether `b` is equal to zero or not. At this point SFE can terminate evaluation with the result:

```
if (b = 0) then 1 else (a * exp a (b - 1))
```

The conditional `if-then-else` is defined using pattern matching. This stopping point is the point at which it cannot determine which case of the definition of `if` to use. The conditional applied to this argument is treated as an uninterpreted function, i.e., as if we do not know its definition. We could continue to evaluate the arguments `((b=0)`, `1`, and `(a * exp a (b - 1)))` further. In this case, continuing to evaluate the arguments indefinitely would result in non-termination. The levels of evaluation of SFE describe when to stop evaluating the arguments of uninterpreted functions. The levels of evaluation are general criteria that are applied across all parts of an expression.

Currently, within a theorem prover, evaluating an expression may require varying amounts of interaction by the user to choose the appropriate sequence of rewriting steps to obtain the desired expansion of the expression. With SFE, we make this expansion systematic using levels of evaluation. Thus, while being less general than rewriting, SFE provides precise control for the user to guide its specialised task. SFE also does not require the unification step needed for rewriting.

There are a variety of applications for symbolic functional evaluation. For example, Boyer and Moore [7] used a more restricted form of symbolic evaluation

[1] A brief explanation of the syntax of S can be found in the appendix.

as a step towards proving theorems in a first-order theory of lists for program verification.

In this paper, we show how SFE allows us to bridge the gap between high-level, expressive requirements notations and automated analysis. The input notations of many specialised analysis tools lack expressibility such as the ability to use uninterpreted types and constants, and parameterisation. One of the benefits of our approach is that it allows specifiers to use these features of higher-order logic in specifications written in notations such as statecharts [21], but still have the benefits of some automated analysis. Therefore, both a novice user and a theorem-proving expert can work on exactly the same specification using different tools. Section 8 provides an example of how general proof results can help the novice user in their analysis of a specification.

We bridge the gap between high-level notations and automated analysis by using SFE as the front-end to more specialised analysis tools. SFE directly evaluates the semantics of a notation in higher-order logic. It produces a representation of the meaning of the specification that can be subjected to behavioural analysis using automated techniques. Our framework is illustrated in Figure 1. SFE plays the role of a \universal translator" from textual representations of notations into expressions of the meaning of the specification.

Because the evaluation process need not always go as far as producing a completely evaluated expression to be sufficient for automated analysis, modes of SFE produce expressions at different levels of evaluation. Different abstractions are then applied for carrying out different kinds of automated analysis. For example, to use BDD-based analysis [8], we can stop evaluation once an uninterpreted function is found at the tip of an expression and then abstract to propositional logic. To use if-lifting, a particular type of rewriting, it is necessary to evaluate partly the arguments of an uninterpreted function.

By directly evaluating the formal semantic functions, our approach is more rigorous than a translator that has not been verified to match the semantics of the notation. Our framework is also more flexible than working in a current theorem proving environment. For example, we are able to return results in terms of the original specification and provide access to the user to control parameters such as BDD variable ordering.

Another application for symbolic functional evaluation is the symbolic simulation step used in many microprocessor verification techniques. Cohn [11], Joyce [26], and Windley [40] all used unfolding of definitions to execute opcodes in theorem proving-based verification efforts. More recently, Greve used symbolic simulation to test microcode [20]. The pipeline flushing approach of Burch and Dill begins with a symbolic simulation step [9]. Symbolic functional evaluation could be used inside or outside of a theorem prover to carry out this step for typed higher-order logic specifications. Inside a theorem prover, it could be used as a \super-duper tactic" [1].

The use of higher-order logic to create a formal specification can easily involve building up a hierarchy of several hundred declarations and definitions, including semantic functions for notations. SFE has been an effective tool in the analysis of

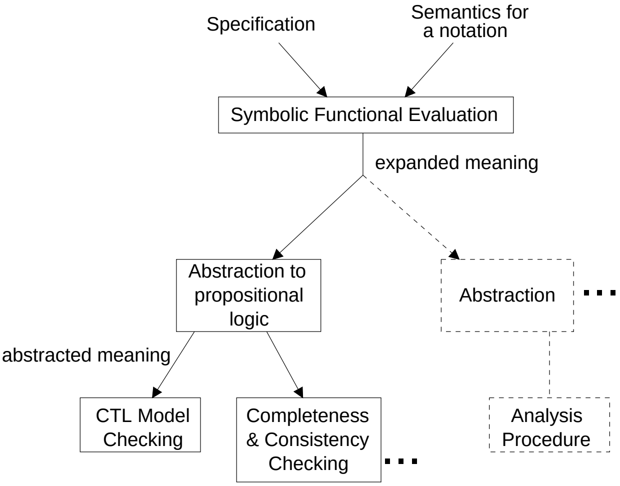


Fig. 1. Our framework

some large specifications. We have used our approach to analyse an aeronautical telecommunications network (ATN) written in a statecharts-variant embedded in higher-order logic. The specification consisted of approximately 3100 lines of text. The SFE step for the ATN took 111 seconds on an Ultra-Sparc 60 (300 MHz) with 1 GB RAM running SunOS 5.6 .

We present our algorithm in terms of expressions in the lambda calculus (applications, abstractions, variables). This presentation provides a simple interface for integrating our implementation of SFE with other tools, as well as giving enough details to implement SFE. SFE makes it possible to have a general-purpose specification notation as a front-end for many specialised analysis tools.

2 Related Work

Translation into the input notation of an existing analysis tool is a common approach to bridge the gap between specialised notations and analysis tools (e.g., [4,5,41]). There are three disadvantages to the translation approach. First, unless the translator has been verified, there is no assurance that the translator correctly implements the semantics of the notation. For notations such as statecharts, establishing the correctness of the translation is very difficult. Second, results are presented to the specifier in the terms of the translated specification, not the original specification. These results can be difficult to decipher. Third, translation often must include an abstraction step because the destination notation is unable to represent non-finite or uninterpreted constants. The translator then matches only a particular type of analysis.

Owre, Rushby, and Shankar have already demonstrated that the approach of embedding a notation in a general-purpose base formalism, such as the PVS form of higher-order logic, makes a range of analysis techniques accessible to a specification [31]. We show that this approach does not require theorem proving support. Besides being difficult to learn for a novice user, theorem provers are verification-based tools and often use decision procedures only for \yes/no" answers. They usually lack the ability to access counterexamples in terms of the original specification and do not allow the user to control the analysis with information such as BDD variable orderings. Our approach is more flexible because it operates independently of the analysis technique, and thus is unencumbered by the effort of dealing with layers of proof management often necessary to integrate decision procedures into theorem provers. This difficulty is noted in the work on integrating the Stanford Validity Checker (SVC) [25] with PVS [32].

It is possible to embed a notation in a functional programming language such as Haskell [33]. This approach has been used for creating domain-specific notations such as the hardware description language Hawk [29]. Symbolic data types can be used to represent uninterpreted constants but this approach requires explicitly describing a symbolic term structure [16]. Also, a programming language lacks a means of representing uninterpreted types and quantification.

There have been efforts to translate higher-order logic specifications into programming languages for execution [2,10,35]. These approaches have been limited to subsets of higher-order logic, often not including uninterpreted constants.

Boyer and Moore [7] used evaluation as a step in proving theorems in a first-order theory of lists for program verification. Their EVAL function is similar to SFE in that it deals with skolem constants. However, SFE also handles uninterpreted function symbols, which raises the question of how much to evaluate the arguments to these symbols. We provide a uniform treatment of this issue using levels of evaluation. These levels work for both uninterpreted functions, and functions such as the conditional defined via pattern matching. If the argument cannot be matched to a pattern, these functions are also treated as uninterpreted. SFE carries out lazy evaluation, whereas EVAL is strict.

Symbolic functional evaluation resembles on-line partial evaluation. The goal of partial evaluation is to produce a more efficient specialised program by carrying out some expansion of definitions based on static inputs [12]. With SFE, our goal is to expand a symbolic expression of interest for verification (i.e., all function calls are in-lined). For this application, all inputs are symbolic and therefore divergence of recursive functions in evaluation often occurs. Our levels of evaluation handle the problem of divergence. We have special treatment of uninterpreted constants in substitution because these are free variables. SFE also handles quantifiers and constants of uninterpreted types.

3 Embedding Notations

Gordon pioneered the technique of embedding notations in higher-order logic in order to study the notations [19]. Subsequent examples include a subset of the programming language SML [39], the process calculus value-passing CCS [30], and the VHDL hardware description language [38]. In previous work [13], we presented a semantics for statecharts in higher-order logic. We use an improved version of these semantics for analysing specifications written in statecharts in our framework. For model-oriented notations, such as statecharts, the embedding of the notation suitable for many types of automated analysis is an operational semantics that creates a next state relation. Symbolic functional evaluation works with both shallow and deep embeddings of notations [6].

4 A Simple Example

For illustration, we choose an example that is written in a simple decision table notation used previously in the context of carrying out completeness and consistency analysis [15]. Table 1 is a decision table describing the vertical separation required between two aircraft in the North Atlantic region. This table is derived from a document that has been used to implement air traffic control software. These tables are similar to AND/OR tables [28] in that each column represents a conjunction of expressions. If the conjunction is true, then the function `VerticalSeparationRequired` returns the value in the last row of the column. Expressions in columns are formed by substituting the row label into the underscore (later represented as a lambda abstraction). In this specification, the functions `FlightLevel`, and `IsSupersonic` are uninterpreted and act on elements of the uninterpreted type `flight`. They are declared as:

```
: flight;               /* declaration of an uninterpreted type */

FlightLevel : flight -> num;    /* declaration of uninterpreted */
IsSupersonic : flight -> bool;         /* constants */
```

We are interested in analysing this table for all instantiations of these functions. For example, we want to know if the table is consistent, i.e., does it indicate different amounts of separation for the same conditions? Specialised tools for carrying out completeness and consistency checking are based on notations that lack the ability to express uninterpreted constants and types (e.g., [22,23]). We use SFE as a universal translator to determine the meaning of a specification in this decision table notation. Using abstraction mechanisms, we can then convert the meaning of the table into a finite state form, suitable for input to automated analysis tools.

Table 1. Vertical separation

					Default
FlightLevel A	_ < = 280	.	_ > 450	_ > 450	
FlightLevel B	.	_ < = 280	_ > 450	_ > 450	
IsSupersonic A	.	.	_ = T	.	
IsSupersonic B	.	.	.	_ = T	
VerticalSeparationRequired(A,B)	1000	1000	4000	4000	2000

We represent tabular specifications in higher-order logic in a manner that allows us to capture the row-by-column structure of the tabular specification. The translation from a graphical representation into a textual representation does not involve interpreting the semantic content of the specification. Table 1 is represented in higher-order logic as:

```
VerticalSeparationRequired (A,B) := Table
[Row (FlightLevel A)  [(\x.x<=280); Dc ; (\x.x>450); (\x.x>450)];
 Row (FlightLevel B)  [Dc; (\x.x<=280); (\x.x>450); (\x.x>450)];
 Row (IsSupersonic A) [Dc; Dc; True; Dc];
 Row (IsSupersonic B) [Dc; Dc; Dc; True] ]
[1000; 1000; 4000; 4000; 2000];
```

`Dc` is \don’t care" replacing the \." in the table. The notation `\x.` is a lambda abstraction. The syntax `[ ... ; ... ]` describes a list. In previous work, we gave a shallow embedding of this decision table notation in higher-order logic by providing definitions for the keywords of the notation such as `Row` and `Table`. We provide these definitions in the appendix.

The result of using SFE to evaluate the semantic definitions for the expression `VerticalSeparationRequired(A, B)` is:

```
if (FlightLevel A <= 280) then 1000
else if (FlightLevel B <= 280) then 1000
else if (FlightLevel A > 450) and (FlightLevel B > 450)
      and (IsSupersonic A) then 4000
else if (FlightLevel A > 450) and (FlightLevel B > 450)
      and (IsSupersonic B) then 4000
else 2000
```

The result of SFE is an expanded version of the meaning of the specification that can be subjected to behavioural analysis techniques. This result is semantically equivalent to the original specification. It is the same as would be produced by unfolding definitions and beta-reduction in a theorem prover, but we accomplish this without theorem proving infrastructure.

An abstracted version of the specification is usually needed for finite state automated analysis techniques. For completeness and consistency analysis, one suitable abstraction is to use a Boolean variable to represent each subexpression that does not contain logical connectives. For example, `FlightLevel B <= 280` is represented as a single Boolean variable. This process of abstracting to propositional logic is based on previous work by Rajan [36], and others. This abstraction is conservative in that the abstract version has more behaviours than the original specification.

5 Symbolic Functional Evaluation (SFE)

Our SFE algorithm is an extension of the spine unwinding algorithm for evaluation of functional programs found in Peyton Jones [34]. Functional programs are essentially the lambda calculus without free variables. Uninterpreted constants do not have definitions and are free variables in the lambda calculus.[2] We extend Peyton Jones' algorithm to include a case for variables with special treatment for the arguments of uninterpreted functions. For efficiency, we make a distinction between uninterpreted constants and other lambda calculus variables. We also introduce evaluation levels. Because the evaluation process need not always go as far as producing a completely evaluated expression to be sufficient for automated analysis, modes of SFE produce expressions at different levels of evaluation.

5.1 Levels of Evaluation

Levels of evaluation are based on the extent to which the arguments to uninterpreted constants, variables and data constructors in an expression are evaluated. The tip of an application is the leaf of the leftmost branch of an application, e.g., `f` for the expression `f a b c` as illustrated in Figure 2. Defined constants and abstractions at the tip are eliminated in evaluation.

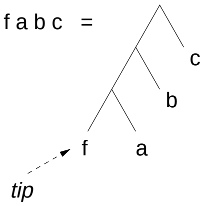


Fig. 2. Tip of a function application

We introduce three levels of evaluation: evaluated to the point of distinction (PD_EVAL), evaluated for rewriting (RW_EVAL), and completely evaluated (SYM_EVAL). These levels are ordered from the \least evaluated" to the \most evaluated". The desired level of evaluation is an input to the symbolic functional evaluation process. The user decides on the level of evaluation based on the type of automated analysis to be carried out. Here we present the levels informally, however a full specification of the levels of evaluation can be found in Day [14].

Evaluated to the Point of Distinction (PD_EVAL) If abstraction to propositional logic is used, evaluated to the point of distinction is sufficient, because the extra information exposed by further evaluation is lost in the abstraction process. For example, an expression such as:

```
IsOnRoute Routes1 A
```

with the declarations and definitions:

```
A : flight;

IsOnRoute : (location # location)set -> flight -> bool;

Routes1 :=
  {(USA,BDA);(CAN,BDA);(IberianPeninsula, Azores);
   (Iceland,Scandinavia);(Iceland, UnitedKingdom)};
```

[2] The G-machine [24], another implementation of graph reduction for evaluating functional programs, is not applicable here because of the free variables. It is necessary to avoid variable capture when doing substitutions of arguments that may contain free variables.

when completely evaluated becomes:

```
        IsOnRoute {(USA,BDA);(CAN,BDA);(IberianPeninsula, Azores);
              (Iceland,Scandinavia);(Iceland, UnitedKingdom)} A
```

If subjected to abstraction to propositional logic, this whole expression is treated as one Boolean variable { expanding the definition of `Routes1` adds information that is lost in the abstraction process when replaced by a single variable. Therefore evaluation to the point of distinction only evaluates an expression to the point where the tip of the expression is an uninterpreted constant or data constructor.

Evaluated for Rewriting (RW_EVAL) To carry out \if-lifting", evaluation to the point of rewriting is needed. If-lifting is a method of rewriting expressions involving the conditional operator `if-then-else` to further reduce the expression. For example, if the value returned by the conditional expression is Boolean, then the following equality holds and can be used to eliminate the `if` function:

```
        if a then b else c ≡ (a and b) or (not(a) and c)        (1)
```

Jones et al. [25] describe \if-lifting" of expressions as a heuristic for their validity checking algorithm. They present two rules:[3]

```
        ((if a then b else c) = (if a then d else e))    ≡
                   if a then (b = d) else (c = e)

  ((if a then b else c) = d)   ≡   if a then (b = d) else (c = d)
```

We generalise these rules slightly to lift an argument with a conditional outside any uninterpreted function (not just equality). Eventually the expression may reach the point where equation (1) can be applied. Transforming an expression by if-lifting and then carrying out abstraction to propositional logic makes the abstraction less conservative. If-lifting can optionally be carried out during symbolic functional evaluation.

Evaluation to the point of rewriting evaluates each argument of an uninterpreted function to the point of distinction. This exposes conditional operators at the tip of the arguments so that if-lifting can be carried out. The `if` operator is used in the semantics of the decision table notation described in the simple example. Therefore, evaluation to the point of rewriting is often chosen for analysing decision tables.

Completely Evaluated (SYM_EVAL) Complete evaluation means all possible evaluation is carried out. The expression may still contain uninterpreted constants as well as some built-in constants of higher-order logic such as conjunction. Complete evaluation can be very helpful in checking the correctness of the semantics. It may also produce more succinct output than either of the other two levels, however, complete evaluation may not terminate.

[3] We use \if-then-else" rather than \ite".

5.2 Algorithm

SFE evaluates expressions in higher-order logic to the point where the expression is at a particular level of evaluation. The user chooses the mode for SFE usually based on the least amount of evaluation that is needed for the type of analysis to be carried out.

Although higher-order logic notation may be enhanced with various constructs, fundamentally it consists of just four kinds of expressions: 1) applications, 2) abstractions, 3) variables, and 4) constants. We subdivide the category of constants into: 4a) uninterpreted constants (including quantifiers), 4b) defined constants, 4c) data constructors, and 4d) built-in constants. Evaluation involves definition expansion, beta-reduction, and evaluation of built-in operations.

Our algorithm carries out normal order reduction, which means arguments to functions are not evaluated until they are used. Evaluation is carried out in place. Figure 3 gives the top-level algorithm in C-like pseudo code. It is called initially with an expression, an empty argument list, and the desired level of evaluation of the expression. In spine unwinding, the arguments to an application are placed on an expression list until the tip of the application is reached.

Compared to Peyton Jones' algorithm, we include extra checks at the beginning to stop evaluation if we have reached the correct level of evaluation. Some subexpressions may already have had some evaluation carried out on them because common subexpressions are represented only once. In this we differ from many interpreters and compilers for functional languages. Evaluation results are also cached.

We also include cases for variables and uninterpreted constants (a sub case of constants). In the cases for variables, uninterpreted constants, and early stopping points in evaluation, we have to recombine an expression with its arguments. Depending on the desired level of evaluation, `Recombine` may carry out more evaluation on the arguments of an uninterpreted function. For example, for evaluated for rewriting, the arguments are evaluated to the point of distinction.

The possible values for the \mode" parameter are elements of the ordered list SYM_EVAL, RW_EVAL, and PD_EVAL. All expressions begin with the level NOT_EVAL. An expression's evaluation tag is stored with the expression and is accessed using the function `EvalLevel`.

If the expression is an abstraction, the arguments are substituted for the parameters in the body of the lambda abstraction avoiding variable capture, and the resulting expression is evaluated. While uninterpreted constants are variables in the lambda calculus, we optimise substitution by treating them differently from other variables. By adding a flag to every expression to indicate if it has any variables that are not uninterpreted constants, we avoid walking over subexpressions that include uninterpreted constants but no parameter variables.

If the expression is an application, spine unwinding is carried out. After evaluation, the original expression is replaced with the evaluated expression, using the function `ReplaceExpr`. A pointer to the expression is used.

The evaluation of constant expressions (`EvalConstant`) is decomposed into cases. If the constant is a constructor or an uninterpreted constant, it is recom-

```
expr EvalExpression(expr *exp, expr_list arglist, level mode)

if (arglist==NULL) and (EvalLevel(exp) >= mode) then
    return exp
else if (EvalLevel(exp) >= mode) and (mode == PD_EVAL)  then
    return Recombine(exp,arglist,NOT_EVAL)

switch (formof(exp))
case VARIABLE (v) :
    if (arglist!=NULL) then return Recombine(exp, arglist, mode)
    else return exp
case ABSTRACTION (parem exp):
    (leftover_args,newexp) = Substitute(exp, parem, arglist)
    return EvalExpression(newexp,leftover_args,mode)
case APPLICATION (f a) :
    newarglist = add a to beginning of arglist
    newexp = EvalExpression(f, newarglist,mode)
    if (arglist==NULL) then ReplaceExpr(exp, newexp)
    return newexp
case CONSTANT(c) :
    return EvalConstant(exp, arglist, mode)
```

Fig. 3. Top-level algorithm for symbolic functional evaluation

bined with its arguments, which are evaluated to the desired level of evaluation. If the constant is a built-in function, the particular algorithm for the built-in constant is executed. If the expression is a constant defined by a non-pattern matching definition, it is treated as an abstraction. For a constant defined by a pattern matching definition, its first argument must be evaluated to the point of distinction and then compared with the constructors determining the possible branches of the definition. If a match is not found, the expression is recombined with its arguments as if it is an uninterpreted constant.

Some built-in constants have special significance. Conjunction, disjunction and negation are never stopping points for evaluation when they are at the tip of an expression because they are understood in all analysis procedures that we have implemented (completeness, consistency, and symmetry checking, model checking, and simulation). Our implementation of SFE can be easily extended with options to recognise particular uninterpreted constants (such as addition) and apply different rules for the level of evaluation of the constant's arguments and do some simplification of expressions. For example, SFE simplifies `1 + a + 1` to `2 + a`, where `a` is an uninterpreted constant.

Experiments using SFE as the first step to automated analysis revealed the value of presenting the user with the unevaluated form of expressions to interpret the results of the analysis. Evaluation in place implies the original expression is no longer available. However, by attaching an extra pointer field allowing the node to serve as a placeholder, the subexpressions of the old expression remain present. An option of SFE keeps the unevaluated versions of expressions.

No special provisions have been taken to check for nontermination of the evaluation process.

6 Quantifiers

Writing specifications in higher-order logic makes it possible to use quantifiers, which rarely appear in input notations for automated analysis tools. In this section, we describe two ways to deal with quantifiers that make information bound within a quantifier more accessible so that less information is lost in abstraction. These logical transformations can be optionally carried out during SFE.

First, we handle quantifiers over enumerated types. A simple enumerated type is one where the constructors do not take any arguments. If the variable of quantification is of a simple enumerated type, then the quantifier is eliminated by applying the inner expression to all possible values of the type. For example, using the definitions,

```
: chocolate := Cadburys | Hersheys | Rogers ; /* type definition */
tastesGood : chocolate -> bool;
```

the expression

```
forall (x:chocolate). tastesGood (x)
```

is evaluated to:

```
tastesGood (Cadburys) and tastesGood (Hersheys) and tastesGood (Rogers)
```

Second, specialisation (or universal instantiation) is a derived inference rule in HOL. Given a term t^0 and a term *forall x:t* used in a negative position in the term (such as in the antecedent of an implication), the quantified term can be replaced by $t[t^0=x]$, where t^0 replaces free occurrences of x in t. Our implementation has a \specialisation" option that carries out universal instantiation for any uninterpreted constants of the type of the quantified variable when universal quantification is encountered in a negative position in evaluation.

7 Abstraction

Symbolic functional evaluation produces an expression describing the meaning of a specification. After the SFE step, some form of abstraction is usually necessary for automated analysis. The abstraction depends on the type of analysis to be performed. We implemented an abstraction to propositional logic technique and represent the abstracted specification as a BDD. Together with SFE, this abstraction mechanism allows us to carry out completeness, consistency, and symmetry checking, model checking, and simulation analysis of specifications written in high-level notations.

When abstracting to propositional logic, subexpressions that do not contain Boolean operations at their tip are converted to fresh Boolean variables. For example, the statement,

```
(FlightLevel A > 450) and (FlightLevel B > 450) and (IsSupersonic A)
```

can be abstracted to,

```
x and y and z
```

with the Boolean variables `x`, `y`, and `z` being substituted for the terms in the original expression, e.g., `x` for `(FlightLevel A > 450)`. This is a conservative abstraction, meaning the abstracted version will have more behaviours than the original specification. In this process, quantifiers are treated as any other uninterpreted constant. To present the output of analysis to the user, the abstraction process is reversed, allowing counterexamples to appear in terms of the input specification.

Along with reversing the abstraction process, the ability to keep the unevaluated versions of expressions during SFE means expressions can be output in their most abstract form. Keeping the unevaluated version of expressions also makes it possible to recognise structures in the specification that can help in choosing an abstraction. For example, the decision table form highlights range partitions for numeric values such as the flight level. In Day, Joyce, and Pelletier [15], we used this structure to create a finite partition of numeric values to produce more accurate analysis output.

Because BDD variable order is critical to the size of the BDD representation of the abstracted specification, we provide a way for the user to control directly this order. The user runs a procedure to determine expressions associated with Boolean variables in abstraction. They can then rearrange this list and provide a new variable order as input. We use a separate tool (Voss [37]) to determine suitable variable orders for our examples.

Other abstraction techniques could certainly be used. For example, we are considering what abstraction would be necessary to link the result of SFE with the decision procedure of the Stanford Validity Checker [25]. SVC handles quantifier-free, first-order logic with uninterpreted constants.

8 Analysing the Semantics of Notations

One of the benefits of our approach is that it allows specifiers to write in higher-order logic but still have the benefits of some automated analysis. Therefore, both a novice user and a theorem-proving expert can work on exactly the same specification using different tools. The theorem-proving expert might prove that certain manipulations that the novice user can do manually are valid with respect to the semantics of the notation. In this section, we give an example of such a manipulation.

We have used our approach to analyse an aeronautical telecommunications network (ATN) written in a statecharts-variant and higher-order logic [3]. The

ATN consists of approximately 680 transitions and 38 basic statechart states, and after abstraction is represented by 395 Boolean variables. The statechart semantics produce a next state relation. In these semantics, Boolean flags representing whether each transition is taken or not are existentially quantified at the top level. Turning the next state relation into a BDD required building the inner BDD and then doing this quantification. We discovered that the inner BDD was too big to build. Therefore we sought to move in the quantification as much as possible to reduce the intermediate sizes of the BDDs. The system was divided into seven concurrent components. We wanted to view the next state relation as the conjunction of the next state relations for each of the concurrent components by pushing the quantification of the transition flags into the component level. Gordon's work on combining theorem proving with BDDs addresses the same problem of reducing the scope of quantifiers to make smaller intermediate BDDs by using rewriting in HOL [18].

We hypothesise the following property of statecharts that could be checked in a theorem prover such as HOL: if the root state is an AND-state and each component state has the properties:

{ no transition within a component is triggered in whole or in part by the event of entering or exiting a state that is not within the component
{ the sets of names modified by actions for each component are disjoint

then:

$$\begin{array}{c}\texttt{Sc (AndState [}st_1;st_2;\ldots st_n\texttt{])}\ step \equiv \\ \texttt{Sc}\ st_1\ step \quad \texttt{and} \quad \texttt{Sc}\ st_2\ step \quad \texttt{and} \quad \ldots \quad \texttt{and} \quad \texttt{Sc}\ st_n\ step\end{array}$$

where Sc is the semantic function producing a next configuration relation, st_x is a component state, and $step$ is a pair of configurations. (We use the term \configuration" to describe a mapping of names to values that is often called a \state".) We have sketched a proof of this property but have not yet mechanised this proof. This property was used with success to allow our tool to build the next state relation for the system and then carry out model checking analysis.

9 Conclusion

This paper has presented symbolic functional evaluation, an algorithm that carries out definition expansion and beta-reduction for expressions in higher-order logic that include uninterpreted constants. We have described the application of SFE to the problem of linking requirements notations with automated analysis. SFE acts as a universal translator that takes semantics of notations and a specification as input and produces the meaning of the specification. To connect a new notation with automated analysis, it is only necessary to write its semantics in higher-order logic. To use a new analysis procedure with existing notations, it is only necessary to provide the appropriate abstraction step from higher-order logic. By linking higher-order logic directly with automated analysis procedures, some queries can be checked of the specification, even if all types of queries cannot be processed automatically.

Our approach is more rigorous than a custom translator, and more flexible than working in a current theorem-proving environment. For example, we are able to return results in terms of the original specification and provide access to the user to control parameters such as BDD variable ordering.

Our algorithm for symbolic functional evaluation extends a spine unwinding algorithm to handle free variables with special considerations for uninterpreted constants. Levels of evaluation provide a systematic description of stopping points in evaluation to tailor the process to particular forms of automated analysis. Compared to an implementation of rewriting, SFE need not search a database for appropriate rewrite rules in a unification step, nor follow branches that are not used in the end result. SFE could be implemented within a theorem prover as a tactic.

Symbolic functional evaluation is applicable to any expression in higher-order logic. We plan to explore how SFE can be used stand-alone for symbolic simulation.

10 Acknowledgments

For helpful discussions and comments on this paper, we thank Byron Cook, Mark Jones, Dick Kieburtz, Jeff Lewis, and John Matthews of OGI, and the reviewers. The first author is supported by Intel, NSF (EIA-98005542), USAF Air Materiel Command (F19628-96-C-0161), and the Natural Science and Engineering Research Council of Canada (NSERC). This paper is based on the first author's PhD research, which was supported by the BC Advanced Systems Institute, Raytheon Systems Canada, and MacDonald Dettwiller.

References

1. M. D. Aagaard, M. E. Leeser, and P. J. Windley. Towards a super duper hardware tactic. In J. J. Joyce and C.-J. H. Seger, editors, *HOL User's Group Workshop.* Springer-Verlag, August 1993.
2. J. H. Andrews. Executing formal specifications by translating to higher order logic programming. In *TPHOLs*, volume 1275 of *LNCS*, pages 17{32. Springer Verlag, 1997.
3. J. H. Andrews, N. A. Day, and J. J. Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *FORTE/PSTV'97*, 1997.
4. J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Soft. Eng.*, 19(1):24{40, January 1993.
5. R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state space exploration. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.
6. R. Boulton et al. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design*, pages 129{156. North-Holland, 1992.

7. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Jour. of the ACM*, 72(1):129{144, January 1975.
8. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677{691, August 1986.
9. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68{79. Springer-Verlag, 1994.
10. A. J. Camilleri. Simulation as an aid to verification using the HOL theorem prover. Technical Report 150, University of Cambridge Computer Laboratory, October 1988.
11. A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127{139, June 1989.
12. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, pages 493{501. ACM Press, 1993.
13. N. Day and J. Joyce. The semantics of statecharts in HOL. In *HOL User's Group Workshop*, volume 78, pages 338{351. Springer-Verlag, 1993.
14. N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, Department of Computer Science, University of British Columbia, 1998.
15. N. A. Day, J. J. Joyce, and G. Pelletier. Formalization and analysis of the separation minima for aircraft in the North Atlantic Region. In *Fourth NASA Langley Formal Methods Workshop*, pages 35{49. NASA Conference Publication 3356, September 1997.
16. N. A. Day, J. R. Lewis, and B. Cook. Symbolic simulation of microprocessor models using type classes in Haskell. To appear in CHARME'99 poster session.
17. M. Gordon and T. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.
18. M. J. C. Gordon. Combining deductive theorem proving with symbolic state enumeration. Transparencies from a presentation, summer 1998.
19. M. J. C. Gordon. Mechanizing programming logics in higher-order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387{439. Springer-Verlag, 1988.
20. D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *FMCAD*, volume 1522 of *LNCS*, pages 321{333. Springer, 1998.
21. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231{274, 1987.
22. M. P. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. on Soft. Eng.*, 22(6):363{377, June 1996.
23. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231{261, July 1996.
24. T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction*, pages 122{32, 1984.
25. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD*, 1995.
26. J. Joyce. *Multi-Level Verification of Microprocessor Based Systems*. PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.
27. J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 285{299, 1994.
28. N. G. Leveson et al. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684{707, September 1994.

29. J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.
30. M. Nesi. Value-passing CCS in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, pages 352{365. LNCS 780, Springer-Verlag, 1993.
31. S. Owre, J. Rushby, and N. Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, April 1996.
32. D. Y. Park, J. U. Skakkebæk, M. P. Heimdahl, B. J. Czerny, , and D. L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP'98*, 1998.
33. J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell*. Yale University, Department of Computer Science, RR-1106, 1997.
34. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall, 1987.
35. P. S. Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In *Higher Order Logic Theorem Proving and its Applications*, pages 527{535. North-Holland, 1993.
36. P. S. Rajan. *Transformations on Data Flow Graphs*. PhD thesis, University of British Columbia, 1995.
37. C.-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, December 1993.
38. J. P. Van Tassel. A formalization of the VHDL simulation cycle. In *Higher Order Logic Theorem Proving and its Applications*, pages 359{374. North-Holland, 1993.
39. M. VanInwegan and E. Gunter. HOL-ML. In *Higher Order Logic Theorem Proving and Its Applications*, pages 61{74. LNCS 780, Springer-Verlag, 1993.
40. P. J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, 1990.
41. P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379{411, October 1993.

A Semantics for the Decision Table Notation

This appendix appeared previously in Day, Joyce, and Pelletier [15].

The S notation is very similar to the syntax for the term language used in the HOL theorem prover. But unlike HOL, S does not involve a meta-language as part of the specification format for declarations and definitions. Instead, the syntax for declarations and definitions is an extension of the syntax used for logical expressions. (In this respect, S more closely resembles Z and other similar formal specification notations.) For example, the symbol `s := f` is used in S for a definition, e.g., `TWO := 2`, in contrast to an assertion, e.g., `TWO = 2`.

Another difference that will likely be noticed by readers familiar with HOL is the explicit type parameterisation of constant declarations and definitions. Type parameters, if any, are given in a parenthesised list which prefixes the rest of the declaration or definition. This is illustrated in the definitions given below by the parameterisation of `EveryAux` by a single type parameter, `ty`.

Many of the definitions shown below are given recursively based on the recursive definition (not shown here) of the polymorphic type `list`. These recursive definitions are given in a pattern matching style (similar to how recursive functions may be defined in Standard ML) with one clause for the `NIL` constructor (i.e., the non-recursive case) and another clause for the `CONS` constructor (i.e., the recursive case). Each clause in this style of S definition is separated by a `|`. The functions `HD` and `TL` are standard library functions for taking the head (i.e., the first element) of a list and the tail (i.e., the rest) of a list respectively.

Type expressions of the form, `:ty1 -> ty2`, are used in the declaration of parameters that are functions from elements of type `ty1` to elements of type `ty2`. Type expressions of the form `:ty1 # ty2` describe tuples. Similarly, type expressions of the form, `:(ty) list`, indicate when a parameter is a list of elements of type `ty`.

Lambda expressions are expressed in S notation as, `\x.E` (where E is an expression).

The semantic definitions for the tabular notation given in the S notation are shown below.

```
(:ty) EveryAux (NIL) (p:ty->bool) := T |
      EveryAux (CONS e tl) p := (p e) and EveryAux tl p;

(:ty) Every (p:ty->bool) l := EveryAux l p;

(:ty) ExistsAux (NIL) (p:ty->bool) := F |
      ExistsAux (CONS e tl) p := (p e) or ExistsAux tl p;

(:ty) Exists (p:ty->bool) l := ExistsAux l p;

(:ty) UNKNOWN : ty;
(:ty)DC := \(x:ty).T;
TRUE := \x. x = T;
FALSE := \x. x = F;

(:ty1) RowAux (CONS (p:ty1->bool) tl) label :=
              CONS (p label) (RowAux tl label) |
       RowAux (NIL) label := NIL;

(:ty) Row label (plist:(A->bool)list) := RowAux plist label;

Columns t := if ((HD t)=NIL) then NIL
             else CONS (Every (HD) t) (Columns (Map t (TL)));

(:ty) TableSemAux (NIL) (retVals:(ty)list) :=
        if (retVals=NIL) then UNKNOWN else (HD retVals) |
      TableSemAux (CONS col colList) retVals :=
        if  col then (HD retVals) else TableSemAux colList (TL retVals);

(:ty) Table t (retVals:(ty)list) := TableSemAux (Columns t) retVals;
```

Author Index

Aagaard, Mark D. 323
Ambler, Simon J. 221

Berghofer, Stefan 19
Bolignano, Dominique 291

Capretta, Venanzio 131
Caprotti, Olga 109
Cohen, Arjeh M. 109
Coupet–Grimal, Solange 91
Crole, Roy L. 221
Curzon, Paul 293

Day, Nancy A. 341

Grobauer, Bernd 273

Harrison, John 113
Hoffmann, Dirk W. 255
Hurd, Joe 311

Jakubiec, Line 91
Jones, Robert B. 323
Joyce, Jeffrey J. 341

Kammüller, Florian 149
Kropf, Thomas 1

Matthews, John 73
Müller, Olaf 273

Paulson, Lawrence C. 149
Pfeifer, Holger 55

Rueß, Harald 55

Santen, Thomas 37
Schneider, Klaus 255
Seger, Carl-Johan H. 323
Staples, Mark 239
Syme, Don 203

Tahar, Sofiéne 293

Völker, Norbert 5

Wenzel, Markus 19, 149, 167

Xiong, Haiyan 293

Zammit, Vincent 185