Preface

This technical report is the Emerging Trends proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007), which was held during 10-13 September in Kaiserslautern, Germany. TPHOLs covers all aspects of theorem proving in higher order logics as well as related topics in theorem proving and verification.

In keeping with longstanding tradition, the Emerging Trends track of TPHOLs 2007 offered a venue for the presentation of work in progress, where researchers invited discussion by means of a brief introductory talk and then discussed their work at a poster session.

Kaiserslautern, Germany, September 2007
Klaus Schneider and Jens Brandt
# Table of Contents

A High Level Reachability Analysis using Multiway Decision Graph in the HOL Theorem Prover .............................................. 1  
  *Sa’ed Abed, Otmane Ait Mohamed, Ghiath Al Sammane*

Propositional Simplification With BDDs and SAT Solvers ................ 18  
  *Hasan Amjad*

Progress Report on Leo-II, an Automatic Theorem Prover for Higher-Order Logic................................................................. 33  
  *Christoph Benzmueller*

Extensible Object-oriented Data Models in Isabelle/HOL ................. 49  
  *Achim D. Brucker, Burkhart Wolff*

A declarative language for the Coq proof assistant ........................ 65  
  *Pierre Corbineau*

Isabelle Theories for Machine Words ....................................... 81  
  *Jeremy Dawson*

An Architecture for Extensible Click’n Prove Interfaces .................. 97  
  *Holger Gast*

Coinductive Proofs for Streams in PVS .................................... 113  
  *Hanne Gottliebsen*

A Code Generator Framework for Isabelle/HOL ............................. 128  
  *Florian Haftmann, Tobias Nipkow*

CSP Revisited .............................................................. 144  
  *Florian Kammueller*

Acyclicity and Finite Linear Extendability: a Formal and Constructive Equivalence ......................................................... 154  
  *Stephane Le Roux*

Verification of Machine Code Implementations of Arithmetic Functions for Cryptography ....................................................... 170  
  *Magnus Myreen, Michael J. C. Gordon*

Translating Haskell to Isabelle ............................................. 178  
  *Paolo Torrini, Christoph Lueh, Christian Maeder, Till Mossakowski*

A Probabilistic Model for Parametric Fairness in Isabelle/HOL ........ 194  
  *Jinshuang Wang, Xingyuan Zhang, Yosen Zhang, Huabing Yang*
Integrating Isabelle/HOL with Specware 

Stephen Westfold
A High Level Reachability Analysis using Multiway Decision Graph in the HOL Theorem Prover

Sa’ed Abed, Otmane Ait Mohamed and Ghiath Al Sammane
Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
{s_abed,ait,sammane}@ece.concordia.ca

Abstract. In this paper, we provide all the necessary infrastructure to define a high level states exploration approach within the HOL theorem prover. While related work has tackled the same problem by representing primitive BDD operations as inference rules added to the core of the theorem prover, we have based our approach on the Multiway Decision Graphs (MDGs). We define canonic MDGs as well-formed directed formulae in HOL. Then, we formalize the basic MDG operations following a deep embedding approach and we derive the correctness proof for each operation. Finally, the high level reachability analysis is implemented as a tactic that uses our MDG theory within HOL.

1 Introduction
Model checking and deductive theorem proving are the two main formal verification approaches of digital systems. It is accepted that both approaches have complementary strengths and weaknesses. Model checking algorithms can automatically decide if a temporal property holds for a finite-state system. They can produce a counterexample when the property does not hold, which can be very important for the reparation of the corresponding error in the implementation under verification or in the specification itself. However, model checking suffers from the state explosion problem when dealing with complex systems. In deductive reasoning, the correctness of a design is formulated as a theorem in a mathematical logic and the proof of the theorem is checked using a general-purpose theorem-prover. This approach can handle complex systems but requires skilled manual guidance for verification and human insight for debugging. Unfortunately, if the property fails to hold, deductive methods do not give counterexample.

Indeed, the combination of the two approaches, states exploration and deductive reasoning promises to overcome the limitations and to enhance the capabilities of each. Our research is originated by this goal. In this paper, we provide all the necessary infrastructure (data structure + algorithms) to define a high level state exploration in the HOL theorem prover. While related work has tackled the same problem by representing primitive Binary Decision Diagrams (BDD) operations [5] as inference rules added to the core of the theorem prover [8], we have based our approach on the Multiway Decision Graphs (MDGs) [6]. MDG
generalizes ROBDD to represent and manipulate a subset of first-order logic formulae which is more suitable for defining model checking inside a theorem prover. With MDGs, a data value is represented by a single variable of an abstract type and operations on data are represented in terms of an uninterpreted functions. Considering MDG instead of BDD will rise the abstraction level of what can be verified using a state exploration within a theorem prover. Therefore, an MDG structure in HOL allows better proof automation for larger datapaths systems.

The paper is organized as follows: first, we define the MDG structure inside the HOL system; in order to be able to construct and manipulate MDG as formulae in HOL. This step implies a formal logic representation for the MDG or what we call: The MDG Syntax. It is based on the Directed Formulae DF: an alternative vision for the MDG in terms of logic and set theory [3]. Subsequently, all the basic operations are built on the top of The MDG Syntax. Then the definition of the MDG operations, following a deep embedding approach in HOL, is associated naturally with a proof of their correctness. Finally, we define an MDG based reachability analysis in HOL as a tactic that uses the MDG theory.

2 Related Work

The closest work, in approach, to our own are those of Gordon [8, 7] and later Amjad [4]. Gordon integrated the verification system BuDDy (BDD package implemented in C) into HOL. The aim of using BuDDy is to get near the performance of C-based model checker, whilst remaining fully expansive, through with a radically extended set of inference rules [7, 8].

In [9], Harrison implemented BDDs inside HOL without making use of external oracle. The BDD algorithms were used by a tautology-checker. However, the performance was thousand times slower than with a BDD engine implemented in C. Harrison mentioned that by re-implementing some of HOL’s primitive rules, the performance could be improved by around ten times.

Amjad [4] demonstrated how BDD based symbolic model checking algorithms for the propositional \( \mu \) – calculus \((L_\mu)\) can be embedded in HOL theorem prover. This approach allows results returned from the model checker to be treated as theorems in HOL. The approach still leaves results reliant on the soundness of the underlying BDD tools. Therefore, the security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound. Our work focusses more on how one can raise the level of assurance by embedding and proving formally the correctness of those operators in HOL.

In fact, while BDDs are widely used in state-exploration methods, they can only represent Boolean formulae. Our work deals with the embedding of MDGs rather than BDDs. The work of Mhamdi and Tahar [11] builds on the MDG-HOL [10] project, but uses a tightly integrated system with the MDG primitives written in ML rather than two tools communicating as in MDG-HOL system. The syntax is partially embedded and the conditions for well-formedness must be respected by the user. By contrast, we provide a complete embedding of the MDG syntax and the conditions could be checked automatically in HOL.

Another major difference between the above work and ours is that it implements the related inference rules for BDD operators in the core of HOL as
un-trusted code, whereas we implement the MDG operations as a trusted code in HOL.

Verification of BDD algorithms has been a subject of active research and many papers offer studies conducted using different proof assistants such that HOL, PVS, Coq and ACL2. These papers try to extend the prover with a certified BDD package to enhance the BDD performance, while still inside a formal proof system. For example, in [14] the correctness of BDD algorithms using Coq has been proved. The goal is to extract a certified algorithms manipulating BDDs in Caml (the implementation language of Coq).

Our work follows the verification of the Boolean manipulating package, but using MDG instead. The correctness of the model checker is then, defined as a tactic, can be achieved after the proof of these algorithms.

3 Multiway Decision Graphs

MDG is a graph representation of a class of quantifier-free and negation-free first-order many sorted formulae. It subsume the class of Bryant’s (ROBDD) [5] while accommodating abstract data and uninterpreted function symbols. It can be seen as a Directed Acyclic Graph (DAG) with one root, whose leaves are labeled by formulae of the logic True (T)[6].

MDGs are canonical representation, which means that an MDG structure has: a fixed node order, no duplicate edges, no redundant nodes, no isomorphic subgraphs, terms concretely reduced that have no concrete subterms other than individual constants, disjoint primary (nodes label) and secondary variables (edges label).

In the rest of this Section, we describe the way we used to represent the decision diagram from graph representation to Directed Formulae DF. Then, we give a brief explanation of the DF and the well-formedness conditions.

3.1 Decision Diagrams: Graph or Formula

Different approaches have been used to formalize decision diagrams either as terms and formulae or as DAGs. The first is a formal logic representation using data type definitions, while the later is a graphical representation using trees and graphs.

Modeling the decision diagram as a decision tree or graph is motivated by reducing memory space and computation time needed to build a BDD: by eliminating redundancy from the canonical representations as described by [13,14]. The main difficulties are caused by data structure sharing and by the side-effects resulted in the computation. The algorithms usually mark the processed nodes or store the results calculated for a subtree or subgraph in a hash-table to avoid recalculation. The definition of such a mechanism is quit complex for automatic reasoning.

On the other hand, modeling the decision diagram as terms and formulae is smoother for proofs and simplify the induction. The experience shows that obtained performance cannot compete (in terms of time and space usage) with BDD libraries written in languages like C. Yet, using terms and formulae avoid users defining the sharing mechanism. Thus, one would not be occupied with identical subterms occurring in the same structure of a term. The work presented in [8,4] is an example of BDD logical representation.
The choice between the two approaches depends on the formalization objectives. If we want to reason about the correctness of the implementation itself, then we need to define decision diagrams as graphs and do sharing of common sub-trees. Clearly this makes the development and the proofs complex. On the other hand, if we are only interested in a high-level view, for the use of induction, then a logical representation is preferred. This is why, we choose the logical representation in terms of Directed Formulae (DF) to model the MDG syntax in HOL.

3.2 Directed Formulae (DF)

Let $F$ be a set of function symbol and $V$ a set of variables. We denote the set of terms freely generated from $F$ and $V$ by $T(F, V)$. The syntax of a Directed Formula is given by the grammar below. The underline is used to differentiate between the concrete and abstract variables.

\[
\begin{align*}
\text{Sort } S &::= S | \bar{S} \\
\text{Abstract Sort } S &::= \alpha | \beta | \gamma | \ldots \\
\text{Concrete Sort } S &::= a | b | c | \ldots \\
\text{Generic Constant } \mathcal{C} &::= a | b | c | \ldots \\
\text{Concrete Constant } \mathcal{C} &::= a | b | c | \ldots \\
\text{Variable } \mathcal{X} &::= V | \bar{V} \\
\text{Abstract Variable } V &::= x | y | z | \ldots \\
\text{Concrete Variable } V &::= x | y | z | \ldots \\
\text{Directed formulae } DF &::= \text{Disj} | \top | \bot \\
\text{Disj} &::= \text{Conj} \lor \text{Disj} | \text{Conj} \\
\text{Conj} &::= \text{Eq} \land \text{Conj} | \text{Eq} \\
\text{Eq} &::= \frac{A = a}{a \in \mathcal{A}} (A \in T(F, V)) \\
&::= \frac{V = \mathcal{C}}{V \in \mathcal{V}} (A \in T(F, \mathcal{X})) \\
\end{align*}
\]

The vocabulary consists of generic constants, concrete constants (individual), abstract variables, concrete variables and function symbols. DF are always disjunction of conjunctions of equations or $\top$ (truth) or $\bot$ (false). The conjunction $\text{Conj}$ is defined to be an equation only $\text{Eq}$ or a conjunction of at least two equations. Atomic formulae are the equations, generated by the clause $\text{Eq}$. The equation can be the equality of concrete term and an individual constant, the equality of a concrete variable and an individual constant, or the equality of an abstract variable and an abstract term.

Given two disjoint sets of variables $U$ and $V$, a Directed Formula of type $U \rightarrow V$ is a formula in Disjunctive Normal Form (DNF). Just as ROBDD must be reduced and ordered, DFs must obey a set of well-formedness conditions given in [6] such that:

1. Each disjunct is a conjunction of equations of the form:
   - $A = a$, where $A$ is a term of concrete sort $\alpha$ containing no variables other than elements of $U$, and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $u = a$, where $u \in (U \cup V)$ is a variable of concrete sort $\alpha$ and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $v = A$, where $v \in V$ is a variable of abstract sort $\alpha$ and $A$ is a term of type $\alpha$ containing no variables other than elements of $U$;
2. In each disjunct, the LHSs of the equations are pairwise distinct; and  
3. Every abstract variable \( v \in V \) appears as the LHS of an equation \( v = A \) in each of the disjuncts. (Note that there need not be an equation \( v = a \) for every concrete variable \( v \in V \).)

3.3 MDG Operations

We give the definitions of MDG basic operations in terms of DF's [6].

**Conjunction Operation:** The conjunction operation takes as inputs two DFs \( P_i, 1 \leq i \leq n \), of types \( U_i \rightarrow V_i \), and produce a DF \( R = \text{Conj} \left( \{ P_i \}_{1 \leq i \leq n} \right) \) of type \( (\bigcup_{1 \leq i \leq n} U_i) \setminus (\bigcup_{1 \leq i \leq n} V_i) \rightarrow (\bigcup_{1 \leq i \leq n} V_i) \) such that:

\[
\models R \Leftrightarrow (\bigwedge_{1 \leq i \leq n} P_i)
\]  

(1)

Note that for \( 1 \leq i < j \leq n \), \( V_i \) and \( V_j \) must not have any abstract variables in common, otherwise the conjunction cannot be computed.

**Relational Product Operation:** The relational product is used for image computation. It takes as inputs two DFs \( P_i, 1 \leq i \leq n \), of types \( U_i \rightarrow V_i \) and a set of variables \( E \) to be existentially quantified, and produce a DF \( R = \text{RelP} \left( \{ P_i \}_{1 \leq i \leq n}, E \right) \) such that:

\[
\models R \Leftrightarrow ((\exists E)(\bigwedge_{1 \leq i \leq n} P_i))
\] 

(2)

The operation computes the conjunction of the \( P_i \) and existentially quantifies the variables in \( E \). For \( 1 \leq i < j \leq n \), \( V_i \) and \( V_j \) must not have any abstract variables in common. The result of computing conjunction and existentially quantification would be a DF of type \( (\bigcup_{1 \leq i \leq n} U_i) \setminus (\bigcup_{1 \leq i \leq n} V_i) \rightarrow ((\bigcup_{1 \leq i \leq n} V_i) \setminus E) \).

**Disjunction Operation:** The disjunction operation is \( n \)-ary. It takes as inputs two DFs \( P_i, 1 \leq i \leq n \), of types \( U_i \rightarrow V_i \), and produce a DF \( R = \text{Disj} \left( \{ P_i \}_{1 \leq i \leq n} \right) \) of type \( (\bigcup_{1 \leq i \leq n} U_i) \rightarrow V \) such that:

\[
\models R \Leftrightarrow (\bigvee_{1 \leq i \leq n} P_i)
\] 

(3)

The operation computes the disjunction of its \( n \) inputs in one pass and note that this operation requires that all the \( P_i, 1 \leq i \leq n \), have the same set of abstract primary variables.

**Pruning By Subsumption:** The pruning by subsumption takes as inputs two DFs \( P \) and \( Q \) of types \( U \rightarrow V_1 \) and \( U \rightarrow V_2 \) respectively, where \( U \) contains only abstract variables that do not participate in the symbol ordering, and produces a DF \( R = \text{PhyS} \left( P, Q \right) \) of type \( U \rightarrow V_1 \) derivable from \( P \) by pruning (i.e. by removing some of disjoints) such that:

\[
\models R \vee (\exists E)Q \Leftrightarrow P \vee (\exists E)Q
\]  

(4)

The disjuncts that are removed from \( P \) are subsumed by \( Q \), hence the name of the algorithm.

Since \( R \) is derivable from \( P \) by pruning, after the formulae represented by \( R \) and \( P \) have been converted to DNF, the disjuncts in the DNF of \( R \) are a subset of those in the DNF of \( P \). Hence \( \models R \Rightarrow P \). And, from (4), it follows tautologically that \( \models P \land \neg (\exists E)Q \Rightarrow R \). Thus we have

\[
\models (P \land \neg (\exists E)Q \Rightarrow R) \land (R \Rightarrow P)
\]

We can then view \( R \) as approximating the logical difference of \( P \) and \( (\exists E)Q \).
4 The MDG Syntax

4.1 DF in HOL

Using HOL recursive datatype, the MDG sort is either concrete or abstract sort. This is embedded using two constructors called \texttt{Abst\_Sort} and \texttt{Conc\_Sort}. The \texttt{Abst\_Sort} takes as argument an abstract sort name of type \texttt{alpha} and the \texttt{Conc\_Sort} takes a concrete sort name and its enumeration of type \texttt{string} as an input argument. This is declared in HOL as follows:

\[
\text{Sort ::= Abst\_Sort of 'alpha | Conc\_Sort of string → string list}
\]

To determine whether the sort is concrete or abstract, we define predicates over the constructor called \texttt{Is\_Abst\_Sort} and \texttt{Is\_Conc\_Sort}.

In the same way, constants and variables are either of concrete or abstract sort. Individual constant can have multiple sorts depending on the enumeration of the sort, while abstract generic constant is identified by its name and its abstract sort. A variable (abstract or concrete) is identified by its name and sort. This is realized by defining a new HOL type as shown below:

\[
\begin{align*}
\text{Ind\_Cons ::= Ind\_Cons of string → 'alpha Sort} \\
\text{Gen\_Cons ::= Gen\_Cons of string → 'alpha Sort} \\
\text{Abst\_Var ::= Abst\_Var of string → 'alpha Sort} \\
\text{Conc\_Var ::= Conc\_Var of string → 'alpha Sort}
\end{align*}
\]

Functions can be either abstract or cross-operators. Cross-functions are those that have at least one abstract argument. Note that concrete variables are not used since they can be eliminated by case splitting:

\[
\begin{align*}
\text{Abst\_Fun ::= Abst\_Fun of string → 'alpha Var list → 'alpha Sort} \\
\text{Cross\_Fun ::= Cross\_Fun of string → 'alpha Var list → 'alpha Sort}
\end{align*}
\]

We have defined a data type \texttt{DF}. The DF can be True or False or a disjunction of conjunction of equations. Then we define the type definition of a directed formula:

\[
\begin{align*}
\text{DF1, DISJ, CONJ1, Eqn, CONJ are distinct constructors and the constructors} \\
\text{EQUAL1, EQUAL2, EQUAL3, EQUAL4, EQUAL5 are used to define an atomic equation.} \\
\text{The type definition package returns a theorem which characterizes the type} \\
\text{DF and allows reasoning about this type. Note that the type is polymorphic in} \\
\text{a sense that the variable could be represented by a string or an integer number} \\
\text{or any user defined type; in our case we have used the string type.}
\end{align*}
\]
From DF to List To simplify the checking of well-formedness conditions and the embedding of MDG operations, we represent a DF as a list having the following format:

\[
\left[
\begin{array}{l}
\text{eq}_1 \\
\text{eq}_2 \\
\vdots \\
\text{eq}_n
\end{array}
\right]
\begin{array}{c}
\text{disjunct}_1 \\
\text{disjunct}_2 \\
\vdots \\
\text{disjunct}_m
\end{array}
\left[
\begin{array}{l}
\left[\text{lhs}_{11};\text{rhs}_{11}\right] \\
\left[\text{lhs}_{1n};\text{rhs}_{1n}\right] \\
\vdots \\
\left[\text{lhs}_{mn};\text{rhs}_{mn}\right]
\end{array}
\right]
\]

where a DF is given as:

\[
\text{DF} = \text{eq}_1 \land \text{eq}_2 \land \cdots \land \text{eq}_n \lor \text{eq}_{21} \land \text{eq}_{22} \land \cdots \land \text{eq}_{2n} \lor \cdots \lor \text{eq}_{m1} \land \text{eq}_{m2} \land \cdots \land \text{eq}_{mn}
\]

Mapping the DF to a list format simplifies our formalization and enables us to automatize MDG operations by using the infrastructure of the predefined List Theory. However, this representation is transparent for the user of the embedded MDG operations later. It is sufficient to input the DF as formulae and the transformations (proved correct) is done automatically.

4.2 Well-formedness Conditions

Since the DF is represented as a list of equations. The embedding of the well-formedness conditions can be defined straightforward by:

- The first condition is satisfied by construction following the Eqn type definition.
- The second condition is embedded as:

\[
\vdash \text{def}\left(\text{Condition}_2 \emptyset = \top \right) \land \\
\text{(Condition}_2 \left(\text{hd}::\text{tl}_1\right) = \text{ALL}_\text{DISTINCT} \text{ hd} \land \text{Condition}_2 \text{tl}_1)
\]

- The embedding of the third condition requires more work and needs an auxiliary function as shown below:

\[
\vdash \text{def}\left(\text{Condition}_3 \left(\text{hd}_1::\text{tl}_1\right) \emptyset = \top \right) \land \\
\text{(Condition}_3 \left(\text{hd}_1::\text{tl}_2\right) = \top \land \\
\text{(Condition}_3 \left(\text{hd}_1::\text{tl}_1\right) \left(\text{hd}_2::\text{tl}_2\right) = \\
\text{Condition}_3 \text{hd}_1 \left(\text{hd}_2::\text{tl}_2\right) \land \text{Condition}_3 \text{tl}_1 \left(\text{hd}_2::\text{tl}_2\right))
\]

\[
\vdash \text{def}\left(\text{Condition}_3 \text{hd}_1 \emptyset = \top \right) \land \\
\text{Condition}_3 \text{hd}_1 \left(\text{hd}_2::\text{tl}_2\right) = \text{IS}_\text{EL} \text{ hd}_1 \text{ hd}_2 \land \text{Condition}_3 \text{hd}_1 \text{tl}_2)
\]

Finally, the predicate Is_Well_Formed_DF is defined as:

\[
\vdash \text{def}\forall \text{df}. \text{Is}_\text{Well}_\text{Formed}_\text{DF} \text{df} = \\
\text{Condition}_2 \left(\text{STRIP}_\text{DF} \text{df}\right) \land \\
\text{Condition}_3 \left(\text{FLAT}(\text{STRIP}_\text{ABS}_\text{DF} \text{df})\right) \left(\text{STRIP}_\text{DF} \text{df}\right)
\]

where STRIP_ABS_DF function extracts the abstract variables of a DF and STRIP_DF extracts the LHS variables of each disjuncts of a DF. We have implemented a HOL tactic to automatize the checking of well-formedness conditions [12].
5 Embedding of the MDG Operations

In fact, HOL provides predefined logical operations that perform conjunction and disjunction of formulae. However, if the inputs of these operations are well-formed DF, outputs will not be necessary well-formed DF. Also, as the DF represent a canonical graph, the variables order must be preserved to satisfy the well-formedness conditions, which is not satisfied when applying HOL operations. Our embedding, is built to answer specifically these concerns. In this Section, we provide a formal embedding of MDG basic operations as well as the proof of their correctness.

However, the proof strategy consists of feeding the same inputs to the logical HOL predefined operations and to the embedded MDG operations. As the output of the embedded MDG operation is well-formed DF, it needs a refinement step to obtain formulae. These formulae will be compared with the output formulae of logical HOL operation. We check the equivalence of both and prove it as a theorem using structural induction and rewriting rules. We describe the conjunction and PbyS operations in detail. The disjunction and RelP operations are similar to the conjunction. The PbyS operation is different and represents the core of the reachability analysis algorithm. The complete source code for the embedding is available in [12].

5.1 The Conjunction Operation

The method for computing the conjunction of two DFs is applicable when the sets of primary variables of the two DFs are disjoint.

The conjunction operation accepts two sets of DFs (df1 and df2) and the order list L of the node label. The detailed algorithm is given in Algorithm 1.

### Algorithm 1 CONJ_ALG (df1, df2, L)

1: if terminal case then
2: return (trivial result);
3: else
4: for (each disjunct ∈ df1) do
5: DF_CONJUNCTION (disj1, df1, df2, L) recursively
6: for (each disjunct ∈ df2) do
7: HD_SUBST (HD_DISJUNCT (disjt1, df1, disjt1, df2, L)) recursively
8: end for
9: append the result of the HD_DISJUNCT;
10: end for
11: append the result of the DF_CONJUNCTION;
12: end if

The algorithm starts with two well-formed DFs and an order list L. The result DF is constructed recursively and ended when a terminal DF is reached (lines 1 and 2). Lines 4 to 11 recursively, applies the conjunction between df1 and df2 (DF_CONJ function). The DF_CONJUNCTION function determines the conjunction of the first disjunct of df1 and df2 as shown in line 5. More details in the embedding can be found in [1].

The HD_DISJUNCT function determines the conjunction between the first disjunct of both DFs (lines 6 to 8). Then, we apply the substitution to be sure that the result is well-formed DF. The substitution is carried out by taking the
disjunct and check the LHS of each equation (primary variable) does not appear in any equations in the RHS (secondary variable) of the same disjunct. If it appears then we apply substitution by replacing its RHS by the other RHS to respect the well formedness conditions. The substitution is carried out using \texttt{HD_SUBST} function. Line 9 recursively append the result and move to the second disjunct of \texttt{df1}. In line 11, the \texttt{DF_CONJUNCTION} function recursively performs the conjunction of the second disjunct of \texttt{df1} with \texttt{df2} and append it to the result. The detailed algorithm describing the \texttt{HD_DISJUNCT} function is given below:

\begin{algorithm}
\textbf{Algorithm 2} \texttt{HD_DISJUNCT} (\texttt{disj1}_\texttt{df1}, \texttt{disj1}_\texttt{df2}, \texttt{L})
\begin{algorithmic}[1]
\STATE 1. if \((\text{position}(\text{LHS}(\text{Eq1}_\text{df1}, \text{L})) = \text{position}(\text{LHS}(\text{Eq1}_\text{df2}, \text{L})))\) \textbf{then}
\STATE 2. \hspace{1em} if \((\text{RHS of both Eqs are equal})\) \textbf{then}
\STATE 3. \hspace{2em} append \text{Eq1} to the result;
\STATE 4. \hspace{2em} call \texttt{HD_DISJUNCT} (\text{tail}(\text{disj1}_\text{df1}), \text{tail}(\text{disj1}_\text{df2}), \text{L});
\STATE 5. \hspace{2em} \textbf{else}
\STATE 6. \hspace{3em} empty the list and quit the \texttt{HD_DISJUNCT};
\STATE 7. \hspace{1em} \textbf{end if}
\STATE 8. \hspace{1em} \textbf{else if} \((\text{position}(\text{LHS}(\text{Eq1}_\text{df1}, \text{L})) < \text{position}(\text{LHS}(\text{Eq1}_\text{df2}, \text{L})))\) \textbf{then}
\STATE 9. \hspace{2em} append \text{Eq1}_\text{df1} to the result;
\STATE 10. \hspace{2em} call \texttt{HD_DISJUNCT} (\text{tail}(\text{disj1}_\text{df1}), \text{disj1}_\text{df2}, \text{L});
\STATE 11. \hspace{1em} \textbf{else}
\STATE 12. \hspace{2em} append \text{Eq1}_\text{df2} to the result;
\STATE 13. \hspace{2em} call \texttt{HD_DISJUNCT} (\text{disj1}_\text{df1}, \text{tail}(\text{disj1}_\text{df2}), \text{L});
\STATE 14. \hspace{1em} \textbf{end if}
\end{algorithmic}
\end{algorithm}

The function tests if the two equations of the two disjuncts have the same order, by checking the position of the head of both equations (lines 1 and 2) using \texttt{position} function. Line 3 adds the equation to the result and move to the next equation, in both disjuncts, and call \texttt{HD_DISJUNCT} recursively (line 4). Otherwise if the head of both equations are equal but the tail (RHS) are not equal, then the result will be empty and we stop and move to the next disjunct in \texttt{df2} (lines 5 and 6). If the first equation of \texttt{df1} comes before \texttt{df2}, then append it to the result and move to the next equation in the same disjunct and repeat the process recursively (lines 8 to 10). Otherwise, if the first equation of \texttt{df2} comes before \texttt{df1}, then append the equation of \texttt{df2} to the result and repeat the process recursively (lines 11 to 13).

Finally, the conjunction operation is embedded in HOL as:

\[
\vdash_{\text{df}} \forall \text{df1 df2 L. CONJ_ALG df1 df2 L =}
\begin{align*}
&\text{if df1 = TRUE then STRIP_DF_list df2} \\
&\text{else if df2 = TRUE then STRIP_DF_list df1} \\
&\text{else if df1 = FALSE then STRIP_DF_list df1} \\
&\text{else if df2 = FALSE then STRIP_DF_list df2} \\
&\text{else TAKE_HD DF_CONJ (STRIP_DF_list df1) (STRIP_DF_list df2) (union (STRIP_Fun df1) (STRIP_Fun df2)) L}
\end{align*}
\]

We prove the correctness of the conjunction operation as shown in Theorem 1. The detailed proof can be found in [1].

**Theorem 1. Conjunction Correctness**

Let \texttt{df1} and \texttt{df2} be well formed DF. Let \texttt{L} be an order list that is equal to the
union of their order lists. Then, the MDG conjunction of df1 and df2 (CONJ_ALG), and HOL logical conjunction of df1 and df2 (CONJ_LOGIC: mapping a list to disjunction of conjunction of equations), are equivalent.

Conjunction Correctness \(\forall df1\ df2. \exists L. \text{Is\_Well\_Formed\_DF}\ df1 \land \text{Is\_Well\_Formed\_DF}\ df2 \land (\text{ORDER\_LIST}\ df1\ df2 = L) \Rightarrow (\text{CONJ\_LOGIC}\ df1\ df2 = \text{DISJ\_LIST}(\text{CONJ\_ALG}\ df1\ df2\ L))\)

Proof. The goal is to prove the equivalence of MDG conjunction and HOL logical conjunction for these DF. The proof uses structural induction on df1 and df2 and rewriting rules. \(\square\)

5.2 The Relational Product (RelP) Operation

The relational product operation is used to compute the sets of states reachable in one transition from one sets of states. It combines conjunction and existential quantification. Thus, to formalize the RelP operation, we are going to use the embedded conjunction operation in the previous subsection. This simplifies the formalization and shows the reusability of our embedding and proof.

The result of the RelP operation is constructed by calling the function EXIST_QUANT, which is responsible on applying the existential quantification over the result of df1 and df2 conjunction with respect to a set of variables:

\[\text{def}\ (\text{EXIST\_QUANT} [] (hd2::tl2) = []) \land (\text{EXIST\_QUANT} (hd1::tl1) [] = (hd1::tl1)) \land (\text{EXIST\_QUANT} (hd1::tl1) (hd2::tl2) = \text{EXIST\_QUANT} (\text{EXIST\_QUANT1} (hd1::tl1) [hd2]) tl2)\]

Because of lack of space, the detailed embedding and the correctness proof can be found in [1]. First, we use the embedding of the conjunction operation explained in Section 4.1 to get the conjunction of two DFs. Then we embed the extra condition regarding the set of variables L2 to be existentially quantified over the result of the conjunction operation:

\[\text{def}\ \forall df1\ df2\ L1\ L2. (\text{RelP\_ALG}\ df1\ df2\ L1\ L2 = \text{EXIST\_QUANT}(\text{CONJ\_ALG}\ df1\ df2\ L1\ L2))\]

5.3 The Disjunction Operation

The operation computes the disjunction of its inputs and requires that all the DFs must have the same set of abstract primary variables.

**Algorithm 3** DISJ_ALG (df1, df2, L)

1: if terminal case then
2: return (trivial result);
3: else if (STRIP_ABS1_DF df1 = STRIP_ABS1_DF df2) then
4: DF\_DISJUNCTION(df1, df2, L)
5: else
6: return empty list;
7: end if

The Algorithm 3 starts with two well-formed DFs and an order list L. The resulted DF is constructed recursively and ended when a terminal DF is reached (lines 1 and 2). Line 3 checks the equality of the abstract variables in both DFs. If they are equal, then (line 4) determines the disjunction of two DFs by calling DF\_DISJUNCTION. Otherwise, the algorithm returns empty list (line 6).

Then the disjunction operation is defined as:
\[ \forall df_1 df_2 L. \text{DISJ}_{\text{ALG}} df_1 df_2 L = \]
\[
\begin{cases}
\text{if} \ (df_1 = \text{TRUE}) \lor (df_2 = \text{TRUE}) \text{ then } [[](\text{"TRUE"}]] \\
\text{else if} \ (df_1 = \text{FALSE}) \land (df_2 = \text{FALSE}) \text{ then } [[](\text{"FALSE"}]] \\
\text{else if} \ df_1 = \text{FALSE} \text{ then } \text{STRIP}_{\text{DF}} \text{list} \ df_2 \\
\text{else if} \ df_2 = \text{FALSE} \text{ then } \text{STRIP}_{\text{DF}} \text{list} \ df_1 \\
\text{else if} \ \text{FLAT}(\text{STRIP}_{\text{ABS}} \text{DF} \ df_1) = \text{FLAT}(\text{STRIP}_{\text{ABS}} \text{DF} \ df_2) \text{ then } \\
\text{UNION}_{\text{HD}} \text{list}(\text{DF}_{\text{DISJUNCTION}}(\text{STRIP}_{\text{DF}} \text{list} \ df_1)) \ (\text{STRIP}_{\text{DF}} \text{list} \ df_2) L) \\
\text{else } [] \end{cases}
\]

Similarly, the detailed embedding and the correctness proof can be found in [1].

5.4 Pruning by Subsumption (PbyS) Operation

The pruning by subsumption operation is used to approximate the difference of sets represented by DFs. Informally, it removes all the paths of a DF \( P \) from another DF \( Q \). In this subsection, we describe the pruning by subsumption operation constraints, its embedding and the correctness proof. The constraints for PbyS requires as inputs two well-formed DFs of types \( U \rightarrow V_1 \) and \( U \rightarrow V_2 \), respectively. Also, an order list \( L \) that represents the union of the two DFs order lists (pre-conditions) is needed. The constraint related to the execution is: the list of variables \( U \) should contain only abstract variables that do not participate in \( L \). The result of the algorithm must be a well-formed DF that represents the pruning by subsumption of \( df_1 \) and \( df_2 \), and of the same type as \( df_1 U \rightarrow V_1 \) (post-condition).

**Algorithm 4 PbyS\_ALG (df1, df2, L)**

1: if terminal case then
2: return (trivial result);
3: else if (STRIP\_ABS\_RHS\_DF df1 = STRIP\_ABS\_RHS\_DF df2) then
4: if (STRIP\_ABS\_RHS\_DF df1 \notin L) then
5: call DF\_PbyS(df1, df2);
6: else
7: return empty list;
8: end if
9: else
10: return empty list;
11: end if

The Algorithm 4 starts with two well formed DFs and order list \( L \). The resulted DF is constructed recursively and ended when a terminal DF is reached (lines 1 and 2). Line 3 checks the equality of both RHS abstract variables of \( df_1 \) and \( df_2 \). If they are equal, then the algorithm checks if those abstract variables are not included in the order list \( L \) using the function IS\_ABS\_IN\_ORDER (line 4). Otherwise, it returns an empty list (line 10). If the condition is satisfied, then the algorithm determines the pruning by subsumption of the two DFs by calling DF\_PbyS function (line 5). Otherwise, the algorithm returns an empty list (line 7). The DF\_PbyS function is defined as given below:
\[\vdash_{def} (DF_{PbyS} \emptyset (hd2::tl2) L3 (hd4::tl4) (hd5::tl5) L = \emptyset) \land \\
(DF_{PbyS} (hd1::tl1) \emptyset L3 (hd4::tl4) (hd5::tl5) L = (hd1::tl1)) \land \\
(DF_{PbyS} (hd1::tl1) (hd2::tl2) L3 (hd4::tl4) (hd5::tl5) L = \\
\text{if } ((\text{FLAT}(hd4::tl4) = \emptyset) = (\text{FLAT}(hd5::tl5) = \emptyset)) \text{ then} \\
DF_{PbyS} (hd1::tl1) (hd2::tl2) [] [] [] L \\
\text{else} \\
PbyS_1 (hd1::tl1) (hd2::tl2) (hd4::tl4) (hd5::tl5) \land \\
(DF_{PbyS} (hd1::tl1) (hd2::tl2) L3 [] [] L = \\
\text{if } (\text{IS_EL} \ hd1 (hd2::tl2)) \text{ then} \\
DF_{PbyS} tl1 (hd2::tl2) [] [] [] L \\
\text{else} \\
hd1 :: DF_{PbyS} tl1 (hd2::tl2) [] [] [] L)\]

The DF_{PbyS} is applied recursively over two disjuncts and has two main cases:

- The top symbol of df1 is not included in the symbols of df2, then df2 will not subsumed df1.
- The top symbol of df1 and df2 are the same or the top symbol of df1 is included in the symbols of df2. We have three cases:

  - The common top symbol is a concrete variable, then its individual constant (RHS) of every equation of df1 must be the same or included in df2, otherwise it will not subsumed by df2.
  - The common top symbol is an abstract variable, then its (RHS) will be either abstract variable, generic constant or abstract function. In this case, df1 will be subsumed by df2 with suitable substitution for the RHS and the arguments of the abstract function as specified in PbyS_{1}. Otherwise it will not subsumed.
  - The common top symbol is cross-operator, then its individual constant (RHS) of every equation of df1 must be the same or included in df2, otherwise it will not subsumed by df2. Note that, the arguments of the the cross-operator might be substituted.

Finally, the pruning by subsumption operation is:

\[\vdash_{def} \forall df1 df2 L. PbyS_{ALG} df1 df2 L = \\
\text{if } (df1 = \text{TRUE}) \text{ then } [["FALSE"]]] \\
\text{else if } (df2 = \text{TRUE}) \text{ then } [["FALSE"]]] \\
\text{else if } (df1 = \text{FALSE}) \text{ then } [["FALSE"]]] \\
\text{else if } (df2 = \text{FALSE}) \text{ then } (\text{STRIP_DF_list} df1) \\
\text{else if } (\text{IS_ABS_IN_ORDER}(\text{FLAT}(\text{STRIP_ABS_RHS_DF} df2))L=[[]) \text{ then} \\
\text{if } (\text{IS_ABS_IN_ORDER}(\text{FLAT}(\text{STRIP_ABS_RHS_DF} df1))L=[[]) \text{ then} \\
DF_{PbyS} (\text{STRIP_DF_list} df1) (\text{STRIP_DF_list} df2) \\
(\text{union} (\text{STRIP_Fun} df1) (\text{STRIP_Fun} df2)) \text{ (refine list df1)} \\
(\text{HD_list_abs}(\text{STRIP_DF_list_abs_list} df1)) \\
(\text{HD_list_abs}(\text{STRIP_DF_list_abs_list} df2)) L \\
\text{else } [] \\
\text{else } []\]

We show here the correctness proof of PbyS operation in Theorem 2.
Theorem 2. Pruning by Subsumption Correctness

Let \( df_1 \) and \( df_2 \) be well formed DF. Let \( L \) be an order list that is equal to the union of their order lists. Then, the MDG disjunction of \( \text{PbyS\_ALG}(df_1, df_2, L) \) and \((\exists\text{\_LIST\_QUANT\_U} df_2)\), is equivalent to the HOL disjunction of \( df_1 \) and \((\exists\text{\_LIST\_U} df_2)\):

\[
\text{Pruning by Subsumption Correctness} \vdash \forall df_1 \ df_2. \ \exists L_1. \ \exists L_2. \\
\text{Is\_Well\_Formed\_DF} df_1 \land \\
\text{Is\_Well\_Formed\_DF} df_2 \land (\text{ORDER\_LIST} df_1 \ df_2 = L_1) \implies \\
(\text{DISJ\_LIST}(\text{STRIP\_DF\_list} df_1) \lor \\
\text{DISJ\_LIST}(\exists\text{\_LIST}(\text{STRIP\_DF\_list} df_2) L_2)) = \\
(\text{DISJ\_LIST}(\text{PbyS\_ALG} df_1 df_2 L_1) \lor \\
\text{DISJ\_LIST}(\exists\text{\_QUANT}(\text{STRIP\_DF\_list} df_2) L_2))
\]

Proof. The proof uses structural induction on \( df_1 \) and \( df_2 \) and rewriting rules. □

This algorithm is used to check whether a set of states is a subset of another set of states. Let \( df_1, df_2 \) be two DFs of type \( U \rightarrow V \), then we say that \( df_1 \) and \( df_2 \) are equivalent DFs if \( \text{PbyS}(df_1, df_2, L) = \text{PbyS}(df_2, df_1, L) \):

\[
\text{Equivalence} \vdash \forall df_1 \ df_2. \ \exists L. \ \text{Is\_Well\_Formed\_DF} df_1 \land \\
\text{Is\_Well\_Formed\_DF} df_2 \land (\text{ORDER\_LIST} df_1 \ df_2 = L) \land \\
(\text{DISJ\_LIST}(\text{PbyS\_ALG} df_1 df_2 L) = \text{DISJ\_LIST}(\text{PbyS\_ALG} df_2 df_1 L)) \\
\implies (df_1 = df_2)
\]

Technical difficulties may raise at this stage. To respect the formal logic of HOL, the formalization of the Directed Formulae in [15] has been modified. The new DF formalization is more suitable for HOL and avoid potential infinite loops. It ensures the reachability analysis termination when it should occur [3]. In fact, applying induction on DF, with these modifications, ameliorate the reasoning with the MDG structure in HOL. This is one of the contributions of our work. Some goals generate cases where the head of an empty list (element) is equal to an empty list. This is justified because we have incomplete definition of some functions and we don’t cover all cases of list arguments. Also, because the number of subgoals generated is big, modifying one definition may change the flow of the proof. Finally, the proof goes through all the definitions of each operation and gives us more confidence about our embedding.

In fact, the conjunction operation has consumed most of the proof preparation effort. Most of the definitions and proofs are reused by the other operations, especially the relational product operation. The embedding of MDG syntax and the verification of MDG operations sums up to 14000 lines of HOL codes. The complexity of the proof is related mainly to the MDG structure, and the recursive definitions of MDG operations.

6 The Reachability Analysis

We show here, the steps to compute the reachability analysis [6] of an abstract state machine using our MDG operations. The important difference is that we are using our embedded DF operators in a high level. At this stage, the proof expert reasons directly in terms of DF, the internal list representation that we have used in the proof of operations is completely encapsulated.
The non-termination problem

Due to the abstract representation and the uninterpreted function symbols, the reachability analysis algorithm may not terminate [6]. Several practical solutions have been proposed to solve the non-termination problem. The authors in [2, 3] related the problem to the nature of the analyzed circuit. Furthermore, they have characterized some mathematical criteria that leads explicitly to the non termination of particular classes of circuits. Thus, we follow a practical consideration for the MDG reachability. Instead of embedding the theory and the algorithms in general, we rather embed the reachability computation of a particular circuit (DF). Our tactic can be applied to any circuit, but cannot prove the general MDG reachability correctness. We illustrate our technique using the MIN-MAX example.

The MIN-MAX Example

We consider the MIN-MAX circuit described in [6]. The MIN-MAX state machine shown in Figure 1 has two input variables \( X = \{ r; x \} \) and three state variables \( Y = \{ c; m; M \} \), where \( r \) and \( c \) are of the Boolean sort \( B \), a concrete sort with enumeration \( \{ 0; 1 \} \), and \( x, m, \) and \( M \) are of an abstract sort \( s \). The outputs coincide with the state variables, i.e. all the state variables are observable and there are no additional output variables.

![Fig. 1. MIN-MAX State Machine](image)

The machine stores in \( m \) and \( M \), respectively, the smallest and the greatest values presented at the input \( x \) since the last reset (\( r = 1 \)). The \( min \) and \( max \) symbols are uninterpreted generic constants of sort \( s \). The DFs of the individual transition relations, for a particular custom symbol order, are shown below:

\[
\begin{align*}
Tr_r & = (((r = 0) \land (n_r = 0)) \lor ((r = 1) \land (n_r = 1))) \\
Tr_m & = (((r = 0) \land (c = 0) \land (n_m = m) \land (leq_Fun(x, m) = 0)) \lor \\
& ((r = 0) \land (c = 0) \land (n_m = x) \land (leq_Fun(x, m) = 1)) \lor \\
& ((r = 0) \land (c = 1) \land (n_m = x)) \\
Tr_M & = (((r = 0) \land (c = 0) \land (n_M = x) \land (leq_Fun(x, M) = 0)) \lor \\
& ((r = 0) \land (c = 0) \land (n_M = M) \land (leq_Fun(x, M) = 1)) \lor \\
& ((r = 0) \land (c = 1) \land (n_M = x)) \lor ((r = 1) \land (n_M = min))
\end{align*}
\]

The DF of the system transition relation \( Tr \) is the conjunction of these individual transition relations. Firstly, we illustrate how the well-formedness conditions are checked. We give partially the definitions for the corresponding MDG syntax:

\[
\begin{align*}
\vdash_{def} \ & \text{bool} = \text{Conc\_Sort } "\text{bool}" \ ["0";"1"] \\
\vdash_{def} \ & \text{wordn} = \text{Abst\_Sort } "\text{wordn}"
\end{align*}
\]
Then the directed formula $Tr$ is defined as:

\[\vdash \text{\texttt{def}}\ Tr = \text{DF1} (\text{DISJ} \ ^{\text{mdg1}} (\text{DISJ} \ ^{\text{mdg2}} (\text{DISJ} \ ^{\text{mdg3}} \ (\text{DISJ} \ ^{\text{mdg4}} (\text{DISJ} \ ^{\text{mdg5}} (\text{CONJ1} \ ^{\text{mdg6}})))))))\]

Applying the predicate $\text{Is\ Well\ Formed\ DF}$ (conversion tactic) on the above directed formula will result true; in the form of theorem.

\[\vdash \text{Is\ Well\ Formed\ DF} \ Tr\]

Secondly, we define one reachability computational step: $\text{Reach\ Step}$. It takes as inputs: the set of input variables $I$, the set of initial states $Q$, the transition relation $Tr$, the state variables to be renamed $Ren$ and the order list $L$.

$\text{Reach\ Step}$ computes the next reachable state by applying successively $\text{Union\ Step}$ which calls $\text{Next\ State}$ and $\text{Frontier\ Step}$. The $\text{Next\ State}$ computes the set of next states reached from a set of given states with respect to the transition relation of the MIN-MAX. The result is obtained using the DF relational product operator $\text{RelP}$ embedded in Section 4.2. The $\text{Frontier\ Step}$ is used to check if all the states reachable by the machine are already visited. Then the $\text{Union\ Step}$ merges the output of $\text{Frontier\ Step}$ with the set of states reached previously using the $\text{PbyS}$ and disjunction operators embedded in Section 4.4 and 4.3, respectively.

The function $\text{RA}_n$ representing the set of states reachable in $n$ or fewer steps is then defined recursively by

\[\vdash \text{\texttt{def}}\ (\text{RA}_n (0) I I_F Q Q_F Tr Tr_A E Ren L R F R A = R) \land \]
\[\vdash \text{\texttt{def}}\ (\text{RA}_n (\text{SUC} n) I I_F Q Q_F Tr Tr_A E Ren L R F R A = )\]
\[\text{Reach\ Step} I I_F\]
\[\text{Frontier\ Step} I I_F Q Q_F Tr A E Ren L\]
\[\text{RA}_n n I I_F Q Q_F Tr F Tr A E Ren L R F R A = R F R A )\]
\[\text{RA}_n n I I_F Q Q_F Tr F Tr A E Ren L R F R A = R F R A ) \]

the variables $(v=I I_F Q Q_F Tr Tr_A In Ren L R F R A)$ are extracted from the initialization step. Then, to compute the set of reachable states we need to compute $\text{RA}_n (n+1) v, \text{RA}_n 1 v, \text{RA}_n 2 v$ etc. Note that the computation of $\text{RA}_n (n+1) v$ needs the computation of $\text{RA}_n n v$.

Then, we define the MDG reachability analysis $\text{Re}_n$ by calling $\text{RA}_n$:

\[\vdash \text{\texttt{def}}\ (\text{Re}_n I Q E Ren L = )\]
\[\text{RA}_n n (\text{STRIP\ DF\ list} I) (\text{STRIP\ Fun} I)\]
\[\text{RA}_n (\text{STRIP\ DF\ list} Q) (\text{STRIP\ Fun} Q)\]
\[\text{RA}_n (\text{STRIP\ DF\ list} Tr)\]
\[\text{RA}_n (\text{STRIP\ Fun} Tr) (\text{HD\ l\ abs}(\text{STRIP\ DF\ list} Tr)) E Ren L\]
\[\text{RA}_n (\text{rep\ list}(\text{STRIP\ DF\ list} Q))\]
\[\text{RA}_n (\text{STRIP\ Fun} Q) (\text{HD\ l\ abs}(\text{STRIP\ DF\ list} Q))\]

$\text{Re}_n$ terminates if we reach a fixpoint characterized by an empty frontier set. That for some particular $n$, say $n=0$, eventually:

\[\text{RA}_n n I I_F Q Q_F Tr Tr_A In Ren L R F R A = \]
\[\text{RA}_n n (n+1) I I_F Q Q_F Tr Tr_A In Ren L R F R A \]

This condition is tested at each stage and raise an exception (fixpoint not yet reached) or return a success (the set of reachable states).

The set of initial states is described by the DF $Q_0$ shown below:
\[ Q_0 = \left[ ((c = 1) \land (m = \text{max}) \land (M = \text{min})) \right] \]

Also the initial reachable states is \( R_0 = Q_0 \). Then, for the first Reach\_Step the reachable states are:

\[ R_1 = \left[ ((c = 0) \land (m = x1) \land (M = x1)) \lor ((c = 1) \land (m = \text{max}) \land (M = \text{min})) \right] \]

we achieve a fixpoint after three Reach\_Step calls and the reachable states at the third iteration (for this example)\( R_2 \):

\[ R_2 = \left[ ((c = 0) \land (m = x1) \land (M = x2)) \lor (\text{leq}\_\text{Fun}(x1, x2) = 0) \lor \\
((c = 0) \land (m = x2) \land (M = x1)) \lor (\text{leq}\_\text{Fun}(x2, x1) = 1) \lor \\
((c = 1) \land (m = \text{max}) \land (M = \text{min})) \right] \]

Finally, we prove the following fixpoint theorem by instantiating the parameters of MIN-MAX:

\[ \text{Fixpoint } \vdash \exists n_0. \ \forall n. \ (n > n_0) \implies (\text{Re}_A\_\text{N} SUC n \ \text{"I " Q0 " Tr " E " Ren " L " Q0 =} \\
\text{Re}_A\_\text{N} \ n \ \text{"I " Q0 " Tr " E " Ren " L " Q0}) \]

The Generalized Tactic The previous reachability analysis can be generalized as a tactic in order to be applied on other circuits. What will change is only the DF and the set of initial states, if we consider the order list is given. Then our tactic (conversion) in its generalized form can be applied to any DF of a circuit. However, the proof of the reachability fixpoint depends on the structure of the circuit and cannot be considered a general solution to the non-termination problem. The tactic encapsulates the following steps:
1. Formalize the circuit in terms of DF.
2. Check for WF conditions.
3. Formalize Reach\_Step.
4. Formalize RA\_n.
5. Prove fixpoint of Re\_An.

The advantage is that we compute the reachable states for only one iteration and then relying on the induction power in HOL we prove reaching a fixpoint. However, this fixpoint may not exist for some particular circuits. Furthermore, finding the induction scheme is not always a trivial step. If the execution path of the circuit is explicitly inductive like for example a circuit that implements the factorial. Then, the inductive variables are identified easily. For most cases, some knowledge of the circuit is needed as the induction is not explicitly identified as shown in MIN-MAX example.

7 Conclusion and Future Work

MDGs have been proposed to extend BDD in the representation of the relations as well as sets of states, in terms of abstract sorts to denote data values and uninterpreted function symbols to denote data operations. We have MDG as formulae in high order logic using the Directed Formula notations. The formalization of the basic MDG operations is built on the top of our MDG syntax. Internally, we have used a list representation for the DF that is more efficient for the embedding and for the correctness proof. The reachability analysis is performed using our platform: we have shown how a fixpoint computation can
be used to prove the existence of such a fixpoint, depending on the DF circuit structure. Here, the proof is planned over the DF as formulae instead of list representation which raises the abstraction during the proof of reachability.

This work is an important step, to define a state exploration algorithm inside an inductive theorem prover; forward to tackle higher level of abstraction. The work can be extended to implement a complete high level model checking in HOL based on our infrastructure. Including the definition of each $L_{MDG}$ related algorithm; as a tactic. Another, future work is to conduct a complete system level case study to measure the capabilities of this approach and to ensure that our approach does not create an unacceptable penalty in terms of the performance of the model checker; due to the additional theorem proving overhead.

References


Propositional Simplification With BDDs and SAT Solvers

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. We show how LCF-style interactive theorem provers might use BDD engines and SAT solvers to perform normalization, simplification of terms and theorems, and assist with interactive proof. The treatment builds on recent work integrating SAT solvers as non-trusted decision procedures for LCF-style theorem provers. We limit ourselves to propositional logic, but briefly note that the results may be lifted to more expressive logics.

1 Introduction

Interactive theorem provers like PVS [21], HOL4 [10] or Isabelle [22] traditionally support rich specification logics. Automation for these logics is however difficult, and proving a non-trivial theorem usually requires manual guidance by an expert user. Automatic proof procedures on the other hand, while often designed for simpler logics, have become increasingly powerful over the past few years. New algorithms, improved heuristics and faster hardware allow interesting theorems to be proved with little or no human interaction, sometimes within seconds.

By integrating automated provers with interactive systems, we can preserve the richness of our specification logic and at the same time increase the degree of automation [24]. However, to ensure that a potential bug in the integration with the external proof procedure does not render the whole system unsound, theorems in LCF-style [8] provers can be derived only through a small fixed kernel of inference rules. Therefore it is not sufficient for the automated prover to return whether a formula is provable, but it must also generate the actual proof, expressed (or expressible) in terms of the interactive system’s inference rules. HOL4, Isabelle and HOL Light are well known LCF-style provers. PVS, on the other hand, is not implemented in an LCF-style manner, except at a very high level. The Coq interactive theorem prover [13] is not technically LCF-style, but follows the “de Bruijn criterion” of verifying proofs using a proof checker that in spirit is much like an LCF-style kernel.

Formal verification is an important application area of interactive theorem proving. Problems in verification can often be reduced to Boolean satisfiability (SAT) and so the performance of an interactive prover on propositional problems may be of significant practical importance. Binary decision diagrams (BDDs) [2] and SAT solvers [5, 19] are powerful proof methods for propositional logic and it is natural to use them as proof engines within interactive provers.
Recent work [6, 29] showed how LCF-style interactive provers could use SAT solvers as non-trusted decision procedures for propositional logic. We now build on this work to show how, for propositional logic, LCF-style interactive theorem provers might use BDD engines and SAT solvers to perform normalisation, simplification of terms and theorems, and assist with interactive proof, without having to trust the external procedures. The treatment is tool independent, and assumes only that the interactive prover is expressive enough to suppose quantification over pure Boolean formulas.

The next section gives a brief account of the relevant aspects of normal forms, BDDs and SAT solvers, to keep the paper self-contained. In §3 and §4, we look at normalisation and simplification respectively. We end with a look at previous related work and some concluding remarks.

2 Preliminaries

We use $\vdash t$ to denote that $t$ is a theorem in the object logic, i.e., the logic of the interactive prover. We reserve the words “iff” and “implies” for logical equivalence and implication in our proofs, and use $\Leftrightarrow$ and $\Rightarrow$ to denote their respective counterparts in the object logic. Quantification binds weaker than $\Leftrightarrow$, which binds weaker than $\Rightarrow$. Propositional truth is denoted by $\top$ and falsity by $\bot$. We use $\equiv$ to denote syntactic equivalence in the object logic. All other notation is standard.

A literal is either an atomic proposition or its negation. A clause is a disjunction of literals. A monomial is a conjunction of literals. Since both conjunction and disjunction are associate-commutative (AC), clauses and monomials can also be interpreted as sets of literals. If a literal occurs in a set, then we abuse notation and assume its underlying proposition also occurs in the set.

2.1 Normal Forms

A term is in disjunctive normal form (DNF) if it is a disjunction of monomials, and in conjunctive normal form (CNF) if it is a conjunction of clauses. Any propositional term $t$ can be transformed into a logically equivalent term in DNF or CNF, denoted by $\text{DNF}(t)$ and $\text{CNF}(t)$ respectively. Again, by AC, $\text{DNF}(t)$ and $\text{CNF}(t)$ can also be interpreted as sets of sets of literals, and we overload the notation accordingly. We will switch back and forth between the term and set interpretations, as convenience dictates.

Computing normal forms is important in automated reasoning for many reasons, most to do with term rewriting theory. For our purposes, they are also important as many proof procedures accept input terms only in some normal form, e.g., resolution based provers use CNF, or have to compute normal forms internally, e.g., some quantifier elimination methods use DNF.

SAT solvers require their input term to be in CNF. Any term can be transformed to CNF, but the result can be exponentially larger than the original
term. To avoid this, definitional CNF [26] introduces extra propositions as placeholders for subterms of the original problem. We use $dCNF(t)$ to denote the definitional CNF of $t$, and note that it can also be interpreted as a set. We will use $dCNF$ extensively in what follows, so we give a short description here.

The best way to understand the basic idea is by example. Consider the term $t$ given by

$$
\neg(((p \Rightarrow q) \Rightarrow p) \Rightarrow p)
$$

The first preprocessing step is to rewrite away the $\Rightarrow$ operators, using the identity $\vdash p \Rightarrow q \iff \neg p \lor q$, and the second preprocessing step is to then push all negations inwards. Having done this, we obtain

$$
((p \land \neg q) \lor p) \land \neg p
$$

Now we proceed bottom up, introducing abbreviations for each subterm in the form of fresh propositions $v_0, v_1$ etc. We set $v_0 \iff p \land \neg q$ and obtain

$$(v_0 \iff p \land \neg q) \land (v_1 \iff v_0 \lor p) \land \neg p$$

and then introduce $v_1$ to get

$$(v_0 \iff p \land \neg q) \land (v_1 \iff v_0 \lor p) \land v_1 \land \neg p$$

at which point it is easy to see that the term can be made into CNF by applying a standard CNF conversion to each abbreviation. We can now see that the negations were moved inwards to avoid needlessly introducing definition variables. There are other possible optimizations but we will stick with this simple procedure for now.

The term $dCNF(t)$ is not logically equivalent to $t$, since there are valuations for the definitional variables $v_i$ that can disrupt an otherwise satisfying assignment to the variables of $t$. However, it is equisatisfiable. This fact is expressed by the theorem

$$
\vdash t \iff \exists \bar{v} \in V.dCNF(t)
$$

where $V$ is the set of all the definitional variables and $\bar{v}$ indicates that the quantification is over all $v \in V$. The definitional CNF procedure can be augmented to produce this theorem automatically. Henceforth, we reserve the identifier prefix $v$ to refer to the definitional variables only. The equivalence that introduces the abbreviation for each $v_i$ will be referred to as the definition of $v_i$, and the right-hand side of the definition, i.e., the term $v_i$ is abbreviating, will be denoted by $\hat{v_i}$.

### 2.2 BDDs

Binary decision diagrams (BDDs) are data structures for representing Boolean formulas and Boolean operations on them. For instance, the BDD of the term $p \Rightarrow q$ is given in Figure 1. Dotted arcs indicate a valuation of $\bot$ and solid arcs a valuation of $\top$ to the parent node. A path from the root to the 1 node indicates
an assignment that makes the formula true, and a path to the 0 node, a falsifying assignment.

The representation can be made canonical by establishing an ordering on the variables, and can be made efficient by removing redundant arcs and nodes. The reduced ordered BDD corresponding to that of Figure 1 is given in Figure 2. When we say BDD, we mean the ordered and reduced version.

BDDs are canonical up to variable ordering, and have efficient counterparts for all Boolean operations, including quantification. In theory, the problem is NP-complete. In practice, BDDs can often achieve very compact representations. We need not say any more about them. The interested reader may consult [1].

It turns out that representing BDDs efficiently in an LCF style prover causes too high a performance penalty (see §5 for details). Therefore, we assume that the results of BDD operations by themselves cannot produce theorems in the object logic.

### 2.3 SAT Solvers

SAT solvers are efficient algorithms for testing Boolean satisfiability. A SAT solver will accept a Boolean term in CNF and return a satisfying assignment to its variables. If the term is unsatisfiable, the SAT solver will return a resolution refutation proof from the clauses of the input CNF term. Not all SAT solvers can produce this proof, but some can [31], and have been integrated with interactive theorem provers with (mostly) a reasonable slowdown [6, 29]. We assume we
have access to such an integration. Thus, the result of a SAT solver can be represented as a theorem in the object logic. This short description is sufficient for our purposes. A tutorial introduction to resolution based SAT solvers is available [18].

3 Normalization

Our first contribution is to term normalization. Normalization means reducing a term to its normal form. This is traditionally done by rewriting with a set of identities. When computing normal forms in LCF-style theorem provers, we further require that the normalization is done by proof, in effect requiring a theorem that the term is logically equivalent to, or in the case of definitional CNF, equisatisfiable with, the obtained normal form. The requirement of proof generation causes a slowdown. Moreover, if we are not careful, a rewrite based transform can generate large normal forms.

We can instead exploit the compact term representation and speed of BDDs and SAT solvers. The solution is straightforward. To generate the DNF of a term \( t \), we build the BDD of \( t \), and then just read off the set of all satisfying assignments. Each assignment is a monomial and the disjunction \( t' \) of all the assignments is thus in DNF. Further, it is satisfiable iff \( t \) is. The required theorem can be obtained by using the SAT solver to check that \( \neg(t \iff t') \) is unsatisfiable, giving \( \vdash t \iff t' \).

Similarly, the CNF of a term can be obtained by reading off all falsifying assignments. The DNF term \( t' \) thus obtained is logically equivalent to \( \neg t \). Then applying negation to \( t' \) and driving all negations inwards to the atoms, we obtain a term that is in CNF, and equivalent to \( t \). Once again, the SAT solver can be used to obtain the required theorem.

For CNF terms generated in this manner, the redundancy removal algorithm in BDDs guarantees that the clauses are subsumption free, i.e., no clause implies another. Similarly, for DNF terms, no satisfying assignment is a subset of another. This contributes towards keeping the normalised terms small.

This might give the impression that any transform on propositional formulas can be implemented by computing the desired result using an efficient external engine and then confirming the result with a SAT solver. While possible in theory, it may not always work in practice. Many such transforms have sub-exponential worst-case or average-case behaviour, and converting the problem to SAT may not help. Also, this approach has a high overhead of external procedure calls, where the interface is often via disk files. If the transform is done several times on small formulas, the overhead may dominate the benefit.

We compared our method of generating normal forms for CNF generation, with the built-in CNF conversion present in the HOL4 theorem prover, on randomly generated propositional terms of various sizes. The results were inconclusive. On even small terms (say, 15 variables and 300 connectives), both methods ran out of memory. This is expected since CNF terms can become exponentially large and interactive theorem provers are not engineered for efficient storage of
large clausal terms. In particular, our procedure ran out of memory during the 
reading off of the CNF from the BDD. On smaller terms, our method was faster 
in general, but not by much. However there were certain cases where it was much 
slower. We put this down to an unfortunate variable ordering for the BDD, since 
as yet we make no effort to find a good one, and to the fact that the SAT solver 
we use [19] is not tuned for random problems.

BDDs do not scale up as well as SAT solvers, in the sense that as the num-
ber of variables increases, the space requirements for storing BDDs can become 
infeasible. In typical interactive proof, users are unlikely to be using such large 
terms. Our aim however is to support better automation, and automatic methods 
may well operate on large terms. It is worthwhile looking for a way to perform 
normalization using SAT solvers only.

The idea (already well known) is to use the SAT solver to generate all satisfy-
ing assignments for a given term $t$. A simple way of doing this is by the addition 
of blocking clauses. The method works as follows:

1. Let $S$ be the set of known satisfying assignments; set $S = \emptyset$
2. Send $t$ to the SAT solver
3. If the SAT solver returns unsatisfiable, return $S$
4. Otherwise we have a satisfying assignment $\sigma$.
5. Set $S$ to $S \cup \{\sigma\}$
6. Form the conjunction of $\neg\sigma$ with the previous input to the solver and send 
   that to the SAT solver
7. Go to step 3

Note that $\sigma$ is a monomial and so $\neg\sigma$ is a clause, so that $t \land \neg\sigma$ is valid 
input to the solver. By adding $\neg\sigma$ to the term, we ensure that the SAT solver 
cannot return satisfying assignments already in $S$. Indeed, the solver will not 
return assignments that are supersets of any known assignments. Subsets may 
be returned, but the set $S$ can be kept irredudant by adapting well known 
techniques [4, 30].

This is easily seen to terminate. By the end, we have all satisfying assign-
ments, and can derive the DNF of $t$ by forming $\bigvee S$. It is not too hard to change 
the steps above so that we can derive CNF instead. This is of course a very 
expensive way to do normalization, and really is only useful for large terms, i.e., 
thousands of variables. A rough estimate of the work required can be arrived at 
by using a stochastic DNF solution counting method (see Chapter 28 of [27]).

Using blocking clauses as above is not the best approach because it forces 
the solver to redo the search from scratch. Many SAT solvers have incremental 
search capability, where the information learnt from previous searches is retained 
and is applicable so long as the new problem term is an extended version of the 
previous one. In fact, specialised algorithms for enumerating all solutions do even 
better [11]. At the moment we are not aware of any integration of these with 
interactive provers.

If anything, the lesson from this work is that, for the generation of large 
normal forms in interactive provers, space complexity is a far more serious prob-
lem than time complexity. The sizes that can be handled are enough for most interactive proof however, so the work in §4.3 for instance, is of practical use.

4 Simplification

If we restrict ourselves to the propositional structure of a term, simplification usually means reducing the size, or the depth, or both, of the term. Normalization, for instance, eliminates depth altogether (modulo AC), and may often result in a smaller term as well. The downside to normalization as a vehicle for simplification is that, by definition, it destroys the structure of the term in question, and with it, any intuition that the human using the theorem prover may have had about the term. Therefore, simplification by rewriting is typically preferred during interactive proof.

Our second contribution is to show how useful simplification can sometimes be accomplished using BDDs and SAT solvers. An important consideration is to do this simplification without the kind of mangling that normalization produces. There are several applicable scenarios, which we now consider.

4.1 Theorems

Suppose we have a propositional term \( t \), and we wish to check whether or not it is a tautology. This can be done by computing \( \text{dCNF}(\neg t) \) and asking a SAT solver if that term is unsatisfiable. If so, let \( V \) be the set of definitional variables appearing in \( \text{dCNF}(\neg t) \), as in §2.1, and we have

\[
\vdash \text{dCNF}(\neg t) \Rightarrow \bot
\]

iff

\[
\vdash \forall \bar{v} \in V. \text{dCNF}(\neg t) \Rightarrow \bot
\]

iff

\[
\vdash (\exists \bar{v} \in V. \text{dCNF}(\neg t) \Rightarrow \bot
\]

iff

\[
\vdash \neg t \Leftrightarrow \bot \text{ by (1)}
\]

and we can conclude \( \vdash t \).

It is rarely the case that every single clause of \( \text{dCNF}(\neg t) \) is used in the SAT solver’s refutation proof. The subset of clauses that does get used is called the unsatisfiable core. There are algorithms that attempt to find smaller cores [7] given a refutation proof. We can use smaller cores to deduce \( \vdash s \) rather than \( \vdash t \), where \( \vdash s \Leftrightarrow t \) but \( s \) is simpler than \( t \). We now show how to construct \( s \).

Suppose we have obtained a core \( D \), so \( \vdash D \Rightarrow \bot \). Now \( D \subseteq \text{dCNF}(\neg t) \) by definition (of a core). If \( D = \text{dCNF}(\neg t) \) then there is no simplification, so \( s \equiv t \).

Otherwise, we impose an ordering relation \( \prec \) on \( V \), such that \( v \prec v' \) iff \( v \) occurs in \( v' \), i.e., in the right-hand side of the definition of \( v' \). Let \( \prec^+ \) be the transitive closure of \( \prec \). Let \( \preceq \) be the transitive closure of \( \preceq \). Let \( \preceq^+ \) be the transitive closure of \( \preceq^+ \). Then \( s \) is constructed as follows:
1. Let \( V' = \{ v \in V \cap \bigcup_{d \in D} d[v'] = \top \} \cap \bigcup_{d \in D} d.v \xrightarrow{+} v \equiv v' \} \)

2. Let \( V'' = \{ v \in V | \exists v' \in V', v \xrightarrow{+} v' \} \)

3. Let \( D' = D \cup \{ c \in dCNF(\neg t) | \exists v \in (V' \cup V''). v \in c \} \). Since \( \vdash D \Rightarrow \bot \), we have \( \vdash D' \Rightarrow \bot \).

4. Let \( D'' = D'[\hat{v}/v] \subseteq V' \cup \top \), so \( \vdash D'' \Rightarrow \bot \) also. In \( D'' \) we also explicitly reverse the CNF expansions of the definitions of \( v' \in V' \).

5. Simplify away the \( v \in V \) in \( D'' \) to obtain \( D''' \) and set \( s \equiv \neg D''' \). At this point \( \vdash \neg s \Rightarrow \bot \).

To elaborate a little, \( V' \) is the set of the maximal (w.r.t. \( \prec \)) \( v \in V \) that occur in \( D \). For each such \( v \), \( D' \) contains all the clauses of \( dCNF(\neg t) \) containing a variable that was \( \xrightarrow{+} v \). \( V'' \) is the set of \( v \in V \) that are strictly below any \( v' \in V' \).

The implicit strategy is to in effect collect together the clauses comprising the definitions of each \( v \in V' \cup V'' \) and reverse the per-abbreviation CNF conversion applied during the definitional CNF computation outlined in \( \S 2.1 \), though we do not explicitly do this reversal except for the \( v' \in V' \). First, we replace each occurrence in \( D' \) of \( v'' \in V'' \) by its definition. Next, each maximal \( v' \in V' \), is replaced by \( \top \). These replacements give us \( D'' \). Now since each \( v'' \in V'' \) has been substituted into its own definition, the clauses corresponding to its definition can be simplified away. Also, since each maximal \( v' \in V' \) occurs only on the left-hand side of its defining equivalence which has been explicitly reconstructed, replacing it by \( \top \) and simplifying converts that equivalence into \( v' \) except that it has been expanded out fully. This gives us \( D''' \), which is a conjunction of single literals and the fully expanded right-hand sides of the definitions of the maximal \( v \in V' \).

Effectively, \( D''' \) is the term after \( \neg t \) was negated and preprocessed but before definitional CNF was applied, less some top-level structure of \( t \). Then we get \( s \) by negating \( D''' \).

Intuitively, each \( v \in V \) represents a subterm of \( t \). Any such \( v \) remaining in the core are clearly pertinent to the truth value of \( t \). Any \( v' \prec v \) also cannot be ignored since each \( v \) is dependent on their definition. Roughly speaking, we collect together all these variables and back-substitute their definitions into the core, in an attempt to resurrect the structure of \( t \) which is implicitly encoded in the definitions. We need to do a little more to better recover the structure, but before we introduce those complications let us first establish the soundness of the basic idea.

Proposition 1. \( \vdash s \Leftrightarrow t \)

Proof The theorem \( \vdash D' \Rightarrow \bot \) at the end of step 3 above is easily seen to be correct. Note that the occurrence of any \( v \in V \) in \( D' \) is implicitly universally quantified, so the substitutions in step 4 are just instantiations, preserving equivalence. Thus step 5 correctly concludes that \( \vdash \neg s \Rightarrow \bot \). Now, we have \( \vdash D \Rightarrow \bot \) from the SAT solver’s refutation proof. Then \( \vdash \neg s \Leftrightarrow D \), using \( \forall t.t \Rightarrow \bot \Leftrightarrow (t \Leftrightarrow \bot) \). Now \( D \subseteq dCNF(\neg t) \), so \( \vdash dCNF(\neg t) \Rightarrow D \). But \( \vdash D \Rightarrow \bot \), so we have \( \vdash dCNF(\neg t) \Leftrightarrow \neg s \) by transitivity of \( \Leftrightarrow \). Finally, \( \vdash \neg t \Leftrightarrow \exists \bar{v} \in V.dCNF(\neg t) \) by the definitional CNF construction, so we get \( \vdash \neg t \Leftrightarrow \exists \bar{v} \in V.\neg s \). No \( v \in V \) occurs in \( s \) since \( V' \cup V'' \subseteq V \) is the set of definitional variables occurring in \( D \).
but these are all substituted away in step 4. So we conclude \( \vdash \neg t \iff \neg s \) and the required result follows. □

Syntactically, \( s \) does not quite follow the structure of \( t \) yet, mainly because we have ignored the preprocessing steps of definitional CNF such as rewriting away \( \Rightarrow \) operators and moving negations inwards. These preserve equivalence however, and can be reversed by storing suitable information for each subterm. Therefore these operations too can be reversed without affecting Proposition 1. The details are uninteresting. We now present an example, before making some concluding remarks. The example is rather contrived, but we want a small example that provokes the “right” behaviour.

Example Let \( t \) be then term \(((p \Rightarrow q) \Rightarrow p) \Rightarrow p) \lor t'\) where \( t' \Rightarrow \bot \) but \( t' \) is complicated enough that it is beyond the ability of the interactive prover’s simplifier to prove that. We further assume that the prover’s native simplifier cannot prove \( \vdash ((p \Rightarrow q) \Rightarrow p) \Rightarrow p \) either.\(^1\) We use the work already done in §2.1 and obtain

\[ \vdash dCNF(\neg t) \iff (v_0 \iff p \land \neg q) \land (v_1 \iff (v_0 \lor p) \land v_1 \land \neg p \land dCNF(\neg t'))\]

We are not interested in what happens to \( t' \), and apply per-abbreviation CNF to get

\[ \vdash dCNF(\neg t) \iff (v_0 \iff p \land \neg q) \land (\neg v_0 \lor \neg q) \land (\neg v_0 \lor p) \land (\neg v_1 \lor v_0 \lor p) \land (v_1 \lor v_0) \land (v_1 \lor \neg p) \land v_1 \land \neg p \land dCNF(\neg t')\]

At this point, \( V = \{v_0, v_1\} \), ignoring the definitional variables occurring in \( dCNF(\neg t') \). All we need to know about the latter is that they are incomparable with any in \( V \) w.r.t. \( \prec \).

Now \( dCNF(\neg t') \) is satisfiable, so the SAT solver is forced to use the rest of \( dCNF(\neg t) \) to show unsatisfiability. The reader may confirm that if \( D = \{\neg v_0 \lor p, \neg v_1 \lor v_0 \lor p, v_1, \neg p\} \) under the set interpretation, then \( D \Rightarrow \bot \) (recall \( D \) is a CNF term). Then \( V' = \{v_1\} \) and \( V'' = \{v_0\} \). Collecting together the needed clauses, we see that

\[ D' = \{v_0 \lor \neg p \lor q, \neg v_0 \lor \neg q, \neg v_0 \lor p, \neg v_1 \lor v_0 \lor p, v_1 \lor \neg v_0, v_1 \lor \neg p, v_1, \neg p\} \]

and of course \( \vdash D' \Rightarrow \bot \). We explicitly reverse the CNF expansion of the definition of \( v_1 \), to obtain

\[ D' = \{v_0 \lor \neg p \lor q, \neg v_0 \lor \neg q, \neg v_0 \lor p, v_1 \iff v_0 \lor p, v_1, \neg p\} \]

\(^1\) If it can, we just replace with a more complex term. It is certainly beyond the HOL4 simplifier, which is our behind-the-scenes test bed.
Now substituting $v_0$ by its definition and $v_1$ by $\top$, and simplifying just enough to remove the $v_i$, we get $\vdash D'' \Rightarrow \bot$ which looks like

$$\vdash (((p \land \neg q) \lor p) \land \neg p) \Rightarrow \bot$$

and finally reversing the preprocessing step (which also adds back the top-level negation that was added when $t$ was negated prior to applying definitional CNF) we get $\vdash \neg s \Rightarrow \bot$ which looks like

$$\vdash \neg(((p \Rightarrow q) \Rightarrow p) \Rightarrow p) \Rightarrow \bot$$

We can now follow the reasoning in the proof above to conclude that

$$\vdash t \iff (((p \Rightarrow q) \Rightarrow p) \Rightarrow p)$$

$\square$

It is clear that this method did simplify $\vdash t$, and that this kind of simplification is likely beyond the reach of the rewriting-based simplifiers currently in use in interactive theorem provers. But by now it should also be suspected that such clean simplification cannot always be achieved. For instance, the current method is crude in the sense that any subterm that is top-level conjunctive (in a general sense) cannot be simplified in this manner, because the SAT solver would then have to prove unsatisfiability of a disjunctive term and so will likely use clauses from both top-level subterms of that subterm. This is not so bad since disjunctions and implications are not affected, and equivalences and conjunctions can be split on their conjunctive structure and proved separately, but it does impose a limit on usability.

We currently make no effort to simplify away unused definitions (i.e., definitions whose definitional variables did not occur in the core) that fall below (w.r.t. $\prec$) a maximal used definition. The intuition suggests this may not be possible, at least not without considering the SAT solver proof directly.

Finally, whenever the set $V'$ is disconnected in the sense that the subterms corresponding to the definitional variables in $V'$ are not connected by some operator in the original term, some of the top-level structure of $t$ is necessarily lost and replaced by a flat conjunctive structure, deteriorating to more or less CNF in the worst case. We plan to address these shortcomings.

### 4.2 Terms

We have shown how the derivation of small unsatisfiable cores can help simplify theorems. As it stands, the method cannot be used to simplify terms that are not theorems, i.e., given a term $t$, prove $\vdash t \iff s$ where $s$ is in some sense a simpler term. However, with a minor adaptation, almost the same process can sometimes succeed in doing so.

This time we convert $t$ to definitional CNF without negating it first. This results in a theorem $\vdash t \iff \exists \overline{v} \in V. dCNF(t)$. Now we build the BDD of $dCNF(t)$ and read off the CNF structure of the BDD as outlined in §3, obtaining a CNF.
term which we shall call $D$. At this point, we use a SAT solver to confirm that
type $dCNF(t) \Leftrightarrow D$.

Intuitively, our hope is that the redundancy removal in the BDD will have
the same effect as finding a smaller unsatisfiable core in the previous section.
However, since $t$ is not a theorem, the results we can achieve are slightly different.

Using the two theorems we have, we calculate as follows

\[ \vdash t \iff \exists \bar{v} \in V. D \text{ by transitivity of } \iff \]
\[ \text{iff } \vdash (\exists \bar{v} \in V. D) \Rightarrow t \]
\[ \text{iff } \vdash \forall \bar{v} \in V. D \Rightarrow t \]

and then instantiate the $v \in V$ (which do not occur in $t$) and reverse construct
the resulting term as in the previous section, to obtain a term $s$. It follows that
\[ \vdash s \Rightarrow t. \]

Now we check using a SAT solver whether $\vdash t \Rightarrow s$. If so, we have $\vdash t \Leftrightarrow s$
and we have successfully simplified $t$. If not, we have failed in the simplification
attempt, but the resulting scenario has an application described in the next
section.

We note in passing that this method can dispense with BDDs altogether by
using the SAT solver based solution enumeration method described towards the
end of §3. As then, this alternative should be considered for large terms only.

4.3 Goal-directed Proof

If the simplification attempt in the previous section fails, we have a term $s$ such
that $\vdash s \Rightarrow t$ and $\vdash \neg(t \Rightarrow s)$. In other words $s$ is a stronger proposition. In
theory, finding a stronger proposition is trivial: $\bot$ is the strongest proposition.
In practice, finding a stronger proposition that retains some of the structure of
the original term finds an obvious application in interactive proof.

In most interactive theorem provers, a proof begins by setting up as a goal
the term that we hope to show is a theorem. Proof then proceeds in a “backwards”
manner, by reducing the goal to simpler subgoals which we hope eventually to
reduce to axioms (or ground rules) of the object logic. The state of a proof is
represented by the outstanding subgoals, each of which can be represented as a
two-sided sequent $\Gamma \vdash t$, where $t$ is the subgoal and $\Gamma$ is the set of assumptions to
that subgoal. This is known as goal-directed proof. Describing this in any detail
will take us too far afield. The interested reader may consult [10] which contains
several examples of this style, or for that matter any tutorial introduction to a
higher-order interactive prover.

One very useful rule of inference that is found in practically every logic is
modus ponens, i.e.,

\[
\frac{\Gamma \vdash p \quad \Delta \vdash p \Rightarrow q}{\Gamma \cup \Delta \vdash q}
\]

in the propositional two-sided incarnation. The backwards equivalent of this rule
is effectively that if the goal is $\Gamma \vdash q$ and we have a theorem $\Delta \vdash p \Rightarrow q$ such
that $\Delta \subseteq \Gamma$, then the goal can be reduced to $\Gamma \vdash p$. 

28 Hasan Amjad
We can think of our theorem \( \vdash s \Rightarrow t \) as \( \emptyset \vdash s \Rightarrow t \). Hence, given a propositional goal \( t \), one way of simplifying it is to find a stronger but simpler term \( s \). Note that if we had succeeded in our simplification attempt in the previous section, \( \vdash t \iff s \) can also be used to reduce \( t \). The point here is that even in the event of failure, the result that we do have is still of some use. In fact, the structure of \( s \) fits nicely into the scheme since it is just a conjunction of subterms of \( t \), and so can be naturally split into subgoals.

5 Related Work

There is a large body of work on the use of BDDs in interactive provers. One of the earliest results combined higher-order logic with BDDs for symbolic trajectory evaluation [14]. A little later, temporal symbolic model checking was done in PVS [23]. These integrations trusted the underlying BDD engines. Around the same time, a serious attempt at using BDDs in an LCF-style manner [12] reported an approximate 100x slowdown. Later, a larger project added BDDs to the Coq theorem prover [28] and reported similar slowdowns, except that the faster programs were themselves extracted by reflection from the Coq representation, and could thus said to have higher assurance. The penalty for checking BDD proofs has thus more or less ensured that BDDs are not used internally by LCF-style theorem provers, in a non-trusted manner. There have of course been trusted integrations of BDDs with LCF-style provers [9], as well as verifications of aspects of BDD algorithms in such provers [17, 20].

This does not rule out the use of BDDs in interactive provers in general. BDDs are used in the ACL2 prover [16] to help with conditional rewriting (a BDD can be thought of as a nested conditional) and for deciding equality on bit vectors (see ACL2 System Documentation). The PVS theorem prover uses BDDs for propositional simplification [3]. This was in fact the inspiration for our work. Roughly speaking, when invoked on a goal with propositional structure, it uses BDDs to obtain the CNF of the goal, which is then used to split the goal into subgoals. Similar functionality can now easily be added to LCF-style provers using the method of §3, since the propositional structure of interactive goals is typically manageable by a BDD.

Integrations of SAT solvers with interactive provers has a shorter history. The integration is trivial for the case where the solver returns a satisfying assignment: we simply substitute the assignments into the input term and check that the resulting ground term evaluates to \( \top \). This can be done efficiently. LCF-style integration of the unsatisfiability case had to wait for the arrival of proof producing SAT solvers [31]. The first such integrations were reported relatively recently [6, 29]. All LCF-style integrations that we know of so far, use the SAT solvers as one-shot decision procedures. Trusted integrations go further, such as the integration of PVS with the Yices Satisfiability-module-theories (SMT) solver. Work on LCF-style integrations with SMT solvers is underway [6].
6 Conclusions

We have shown how BDDs and SAT solvers can be used for fast normalisation, simplification of terms and theorems, and assistance with interactive proof. Even though we have restricted ourselves to propositional logic, the results can be applied to the propositional structure of more expressive logics via Skolemization, as is done in PVS. The results can also be extended directly to use SMT solvers [25] rather than SAT solvers, using the alternative solutions that avoid BDDs, and should allow us to do simplification in combinations of decidable theories.

Using BDDs to generate normal forms and checking the result is an obvious next step once a non-trusted proof producing SAT solver is available. To the best of our knowledge however, exploiting the non-occurrence of definitional variables in the results of BDDs and SAT solvers for the purposes of simplification, has not been done before. Perhaps because of this, the treatment has an ad hoc feel to it, and many opportunities for optimization exist.

We plan to use SAT solvers and BDDs in a more fine grained manner, perhaps in conjunction with the rewriting system of the theorem prover, as is done in ACL2. We expect to customize SAT solvers and unsatisfiable core finders, for instance to give special treatment to definitional variables. We also hope to generate proofs directly from BDDs, possibly using some of the techniques of [15]. These plans will form the initial steps for future research.

References


Progress Report on Leo-II, an Automatic Theorem Prover for Higher-Order Logic

Christoph Benzmüller\textsuperscript{1,2}, Larry Paulson\textsuperscript{1}, Frank Theiß\textsuperscript{2}, and Arnaud Fietzke\textsuperscript{2}

\textsuperscript{1}Computer Laboratory, The University of Cambridge, UK
\textsuperscript{2}Computer Science, Saarland University, Saarbrücken, Germany

Abstract. Leo-II, a resolution based theorem prover for classical higher-order logic, is currently being developed in a one year research project at the University of Cambridge, UK, with support from Saarland University, Germany. We report on the current stage of development of Leo-II. In particular, we sketch some main aspects of Leo-II’s automated proof search procedure, discuss its cooperation with first-order specialist provers, show that Leo-II is also an interactive proof assistant, and explain its shared term data structure and its term indexing mechanism.

1 Introduction

Automatic theorem provers (ATPs) based on the resolution principle, such as Vampire \cite{vampire}, E \cite{e}, and SPASS \cite{sPASS}, have reached a high degree of sophistication. They can often find long proofs even for problems having thousands of axioms. However, they are limited to first-order logic. Higher-order logic extends first-order logic with lambda notation for functions and with function and predicate variables. It supports reasoning in set theory, using the obvious representation of sets by predicates. Higher-order logic is a natural language for expressing mathematics, and it is also ideal for formal verification. Moving from first-order to higher-order logic requires a more complicated proof calculus, but it often allows much simpler problem statements. Higher-order logic’s built-in support for functions and sets often leads to shorter proofs. Conversely, elementary identities (such as the distributive law for union and intersection) turn into difficult problems when expressed in first-order form.

The LEO-II project develops a standalone, resolution-based higher-order theorem prover that is designed for fruitful cooperation with specialist provers for first-order and propositional logic. The idea is to combine the strengths of the different systems. On the other hand, LEO-II itself, as an external reasoner, wants to support interactive proof assistants such as Isabelle/HOL \cite{isabelle}, HOL \cite{hol}, and OMEGA \cite{omega} by efficiently automating subproblems and thereby reducing user effort.

Leo-II predominantly addresses higher-order aspects in its reasoning process with the aim to quickly remove higher-order clauses from the search space and

\* The Leo-II project is funded by EPSRC under grant EP/D070511/1, “LEO II: An Effective Higher-Order Theorem Prover.”
to turn them into essentially first-order clauses which can then be refuted with a first-order prover. Furthermore, the project investigates whether techniques that have proved very successful in automated first-order theorem proving, such as shared data structures and term indexing, can be lifted to the higher-order setting. Leo-II is implemented in OCAML; it is the successor of LEO [7], which was implemented in LISP and hardwired to the OMEGA proof assistant.

This paper is structured as follows: Sec. 2 presents some preliminaries. Sec. 3 illustrates Leo-II’s main proof search procedure which is based on extensional higher-order resolution. The cooperation of Leo-II with other specialist provers is discussed in Sec. 4. Leo-II is also an interactive proof assistant as we will explain in Sec. 5. In Sec. 6 we address Leo-II’s shared term data structures and term indexing. Sec. 7 mentions related work and Sec. 8 concludes the paper.

2 Preliminaries

Leo-II’s logic Leo-II’s logic is classical higher-order logic (Church’s simple type theory [9]), which is a logic built on top of the simply typed \( \lambda \)-calculus. The set of simple types \( T \) is usually freely generated from basic types \( o \) and \( \iota \) using the function type constructor \( \rightarrow \). In Leo-II we allow an arbitrary but fixed number of additional base types to be specified.

For formulae we start with a set of (typed) variables (denoted by \( X_\alpha, Y, Z, X_1^\beta, X_2^\gamma \ldots \)) and a set of (typed) constants (denoted by \( c_\alpha, f_\alpha \rightarrow \beta \ldots \)). The set of constants includes Leo-II’s primitive logical connectives \( \neg o \rightarrow o \), \( \forall o \rightarrow o \rightarrow o \) and \( \Pi (\alpha \rightarrow o) \rightarrow o \) (abbreviated \( \Pi^\alpha \)) and \( =_{\alpha \rightarrow \alpha \rightarrow o} \) (abbreviated \( =^\alpha \)) for each type \( \alpha \). Other logical connectives can be defined in Leo-II in terms of the these primitive ones (as we will later see).

Formulae (or terms) are constructed from typed variables and constants using application and \( \lambda \)-abstraction. We use Church’s dot notation so that \( \lambda \) stands for a (missing) left bracket whose mate is as far to the right as possible (consistent with given brackets). We use infix notation \( A \vee B \) for \( ((\forall A)B) \) and binder notation \( \forall X_\alpha A \) for \( (\Pi^\alpha (\lambda X_\alpha A_\alpha)) \).

The target semantics for Leo-II in the first place is Henkin semantics; see [6, 15] for further details. Thus, in theory Leo-II aims at a Henkin complete calculus which includes Boolean and functional extensionality as required, for instance, to prove

\[
=^\iota =^\iota =_{\iota \rightarrow \iota \rightarrow o} (\lambda X_\iota. \lambda Y_\iota. Y =^\iota X)
\]

In practice, however, we sacrifice completeness and instead put an emphasize on coverage of the problems we are interested in.

Literals, Unification Constraints, and Clauses Here are examples of literals (\( \alpha \) is an arbitrary type); they consist of a literal atom in [.]-brackets and a polarity \( T \) or \( F \) for positive or negative literal respectively:

\[
[\!A_\alpha\!] =^T \quad [\!B_\alpha\!] =^F \quad [\!C_\alpha =^o D_\alpha\!] =^T \quad [\!F_\alpha =^o G_\alpha\!] =^F
\]
Negative equation literals, such as our fourth example above, are also called a unification constraints. If both terms $F_\alpha$ and $G_\alpha$ in a unification constraint have a free variable at head position then we call it a flex-flex unification constraint. If only one of them has a free variable at head position we call it a flex-rigid unification constraint. Flex-flex and flex-rigid unification constraints may have infinitely many different solutions. For example, the following flex-rigid constraint ($H$ is a variable and $f$ and $a$ are constants) has $H \leftarrow \lambda x. f(f \ldots (f x) \ldots)$ for all $n \geq 0$ amongst its solutions:

\[
[H_{1 \leftarrow \alpha}(f_{1 \leftarrow \alpha}a) = f_{1 \leftarrow \alpha}(H_{1 \leftarrow \alpha}a)]^=F
\]

The following Leo-II clause consists of a positive literal and a unification constraint:

\[
C : [A_\alpha]^=T, [F_\alpha =^\alpha G_\alpha]^=F
\]

It corresponds to the formula

\[
\neg (F_\alpha =^\alpha G_\alpha) \Rightarrow A_\alpha
\]

which explains the name unification constraint: if we can equalize $F_\alpha$ and $G_\alpha$, for example, by unification with a unifier $\sigma$, then $\sigma(A_\alpha)$ holds.

### An Example Problem in Leo-II’s Input Syntax

Leo-II employs a fragment of the new higher-order TPTP THF[1] syntax as its input and output syntax. We present an example problem in THF syntax in Fig. 1. It states that there exists a relation $R$ (of type $(\iota \rightarrow \iota) \rightarrow o$) which is not an equivalence relation. In the notation of this paper this problem reads as follows:

\[
\begin{align*}
\text{reflexive} & \overset{\text{def}}{=} \lambda R_{(i \rightarrow o) \rightarrow o} \forall X_{\iota}(R X X) & (1) \\
\text{symmetric} & \overset{\text{def}}{=} \lambda R_{(i \rightarrow o) \rightarrow o} \forall X_{\iota} \forall Y_{\iota}(R X Y) \Rightarrow (R Y X) & (2) \\
\text{transitive} & \overset{\text{def}}{=} \lambda R_{(i \rightarrow o) \rightarrow o} (\text{reflexive } R) \land (\text{symmetric } R) \land (\text{transitive } R) & (3) \\
\text{equiv_rel} & \overset{\text{def}}{=} \lambda R_{(i \rightarrow o) \rightarrow o} \forall X_{\iota} \forall Y_{\iota} \forall Z_{\iota}((R X Y) \land (R Y Z)) \Rightarrow (R X Z) & (4) \\
\exists R_{(i \rightarrow o) \rightarrow o} & \neg (\text{equiv_rel } R) & (5)
\end{align*}
\]

(1)–(4) are examples of definitions and (5) is the simple higher-order theorem which we want to prove. For this, Leo-II needs to generate a concrete instance for $R$ and show that this instance is not an equivalence relation. Two obvious candidates are inequality or the empty relation:

\[
\begin{align*}
\{(x, y) | x \neq y\} & \text{ represented by } \lambda X_{\iota} \lambda Y_{\iota} \neg (X = Y) & (6) \\
\{(x, y) | \text{false}\} & \text{ represented by } \lambda X_{\iota} \lambda Y_{\iota} \bot & (7)
\end{align*}
\]

As we will see later, Leo-II finds the latter instance and then shows that it does not fulfill the reflexivity property.
\[
\text{thf}(\text{reflexiv}, \text{definition},
\quad (\\text{reflexiv} := (\forall [R: i > (i > o)] : (! [X: i]: ((R @ X) @ X))))).
\]
\[
\text{thf}(\text{symmetric}, \text{definition},
\quad (\text{symmetric} := (\forall [R: i > (i > o)] : (! [X: i, Y: i]: ((R @ X) @ Y) \Rightarrow ((R @ Y) @ X))))).
\]
\[
\text{thf}(\text{transitive}, \text{definition},
\quad (\text{transitive} := (\forall [R: i > (i > o)] : (! [X: i, Y: i, Z: i]: (((R @ X) @ Y) & ((R @ Y) @ Z)) \Rightarrow ((R @ X) @ Z))))).
\]
\[
\text{thf}(\text{equiv_rel}, \text{definition},
\quad (\text{equiv_rel} := (\forall [R: i > (i > o)] : (\text{reflexive @ R}) & (\text{symmetric @ R}) & (\text{transitive @ R}))))).
\]
\[
\text{thf}(\text{test}, \text{theorem},
\quad (\forall [R: i > (i > o)] : \neg (\text{equiv_rel @ R})).
\]

Fig. 1. Example problem in THF syntax.

A Note on Polymorphism

Polymorphism and type inference is supported so far only partially in the Leo-II prover. Full support would add another dimension of complexity and non-determinism, as we will now briefly explain. Consider the following two operation tables

\[
\begin{array}{c|c|c|c}
\hline
\wedge & o & o & o \\
\hline
\top & o & o & o \\
\hline
\bot & o & o & o \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c}
\hline
\wedge & o & o & o \\
\hline
\top & o & o & o \\
\hline
\bot & o & o & o \\
\hline
\end{array}
\]

They are concrete instances of the following polymorphic (or schematic) table

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
\circ & \alpha & \to & \alpha & \to & \alpha \\
\hline
\top & \alpha & \circ & \alpha & \circ & \alpha \\
\hline
\bot & \alpha & \circ & \alpha & \circ & \alpha \\
\hline
\end{array}
\]

(\alpha \neq \beta)

Using type variable \( \alpha \), we can easily formulate a theorem in polymorphic higher-order logic, expressing that there exist an instance of the latter table:

\[
\exists \alpha. \exists \circ_{\alpha \rightarrow \circ_{\alpha \rightarrow \circ_{\alpha}}} \exists A_{\alpha}. \exists B_{\alpha}.
\]
An example is the following modified definition of reflexive, which generalizes from type $i$ to the type variable $A$:

reflexive := $\forall [R:A>(A>\text{type})]: (\forall [X:A]: ((R @ X) @ X))$

3 The Automatic Proof Search Procedure

We sketch some main aspects of Leo-II’s automated proof search procedure.

Problem Initialization

Given a problem in THF syntax consisting of a set of $n \geq 0$ assumptions $A_1, \ldots, A_n$ and a conjecture $C$. Initialization in Leo-II creates the following initial clauses (usually they are not in clause normal form):

$$B_1 : [A_1]^=T \ldots B_n : [A_n]^=T \rightarrow B_{n+1} : [C]^=F$$

For our example problem in (1)–(5) we obtain the following clause

$$C_1 : [\exists R_{(i\rightarrow i)}\rightarrow o \neg (\text{equiv rel } R)]^=F$$

Definition Unfolding

A definition in Leo-II is of form $A \overset{\text{def}}{=} B$ where $\text{Free}(B) \subseteq \text{Free}(A)$. We have already seen simple definition examples in (1)–(4). In particular, non-primitive logical connectives are introduced as definitions in Leo-II; for example:

$$\land_o \rightarrow_o \rightarrow_o \overset{\text{def}}{=} \lambda X_o \lambda Y_o. (X \land Y) \land_o \rightarrow_o \rightarrow_o \overset{\text{def}}{=} \lambda X_o \lambda Y_o. (\neg X \land Y) \land_o \rightarrow_o \rightarrow_o \overset{\text{def}}{=} \lambda X_o \lambda Y_o. (X \supset Y) \land (Y \supset X) \rightarrow \forall X_o. A_o \overset{\text{def}}{=} \Pi^\forall X_o A \exists X_o. A_o \overset{\text{def}}{=} \neg \forall X_o. \neg A$$

Currently, Leo-II simultaneously unfolds all definitions immediately after problem initialization (and thereby heavily benefits from the shared term data structures and indexing techniques as will be sketched in Sec. 6). Delayed and stepwise definition unfolding, which is required to successfully prove certain theorems [8], is future work. When applied to our example clause $C_1$, definition unfolding generates clause 3 as depicted in Fig. 2.

Clause Normalization

For clause normalization, Leo-II employs rules addressing the primitive logical connectives $\neg$, $\lor$, $\Pi^\alpha$ and $=^\alpha$ for all types $\alpha$, as we discuss elsewhere [5]. Clause normalization is an integral part of Leo-II’s calculus and it is not just applied in an initial phase such as in first-order theorem proving. Quite to the contrary, clause normalization is repeatedly required as we will later see.

Unfortunately, clause normalization as currently realized in Leo-II is quite naive. Future work therefore includes the development of a more efficient approach, for example, one based on the ideas of Flotter [29].
Fig. 2. Excerpt of the Leo-II proof protocol for our running example. Clause 13, 25, and 31 are the results of problem initialization, unfolding of definitions, and exhaustive clause normalization. The $V_i$ are free variables.

Fig. 2 shows the result of our running example after problem initialization, definition unfolding, and clause normalization. The clauses present in Leo-II’s search state then are 13, 25, and 31. In our standard notation they read as follows (the $V_i$ are all free variables):

$$
\begin{align*}
C_{15} : & [V^1 V^2 V^2] = T \\
\end{align*}
$$

**Extensional Pre-Unification** Pre-unification in Leo-II is based on the rules as presented in former work [5]. It is well known, that pre-unification is not decidable (remember or discussion of flex-rigid unification constraints in the beginning) which is why Leo-II introduces a depth limit for the generation of pre-unifiers. This clearly threatens completeness. Iterative deepening or dovetailing the generation of pre-unifiers with the overall proof search are possible ways out, however, so far we sacrifice completeness.

Leo-II also provides a unification rule for Boolean extensionality which transforms unification constraints between terms of type $o$ back into proof problems. Consider, for example, clause $D_1$ below, which consists of exactly one unification constraint (a negated equation between two syntactically non-unifiable abstractions). Note how $D_1$ is translated by functional and Boolean extensionality into the propositional-like clauses $D_3$ and $D_4$ ($p, q$ are constants of type $\iota \rightarrow o$ and $s$ is a constant of type $\iota$):

$$
D_1 : [(\lambda X_\iota (pX) \land (qX)) = (\lambda X_\iota (qX) \land (pX))] = F
$$

$$
Func \ D_1 : \quad \begin{align*}
D_2 : & [(p(s) \land (q(s)) = (q(s) \land (p(s))] = F \\
D_3 : & [(p(s) \land (q(s))] = F, [(q(s) \land (p(s))] = F
\end{align*}
$$

$$
Bool \ D_2 : \quad \begin{align*}
D_4 : & [(p(s) \land (q(s))] = T, [(q(s) \land (p(s))] = T
\end{align*}
$$
LEO-II> prove
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 [1] 33 34
35 36 37 38 39 40
Eureka --- Thanks to Corina!
Here are the empty clauses
[
41:[0:<false = $true>-w(1)-i() ]-mln(1)-w(1)-i(sim 35)-fv([ ])
]
0.01373: Total Reasoning Time
LEO-II (Proof Found!)> show-derivation 41
**** Beginning of derivation protocol ****
[...
15: ((V_x0_1 @ V_x1_2) @ V_x1_2)=$true --- cnf 13
35: ($false)=$true --- prim-subst (V_x0_1-->lambda [V25]: lambda [V26]: false) 15
41: ($false)=$true --- sim 35
**** End of derivation protocol ****
LEO-II (Proof Found!)>

Fig. 3. Second excerpt of the Leo-II proof protocol for our running example; see also Fig. 2. From Clause 13 (reflexivity) we derive a contradiction (the clause 41). The key step is the guessing of an appropriate relation via primitive substitution in clause 35.

This example also illustrates why clause normalization is not only a preliminary phase in Leo-II: \( \mathcal{D}_3 \) and \( \mathcal{D}_4 \) are non-normal clauses generated by Leo-II’s extensional pre-unification approach and they need to be subsequently normalized. Note also how the interplay of functional and Boolean extensionality turns our initially higher-order problem (semantic unifiability of two abstractions) into a propositional one. This aspect is picked up again in Sec. 4, where we address cooperation of Leo-II with specialist provers for fragments of higher-order logic, such as first-order and propositional logic.

Resolution, Factorization, and Primitive Substitution
Resolution, factorization and primitive substitution are driving the main proof search of Leo-II. Respective rules are presented in [5]. Unlike in first-order theorem proving, resolution and factorization do not employ unification directly as a filter, but instead introduce unification constraints. These unification constraints are processed subsequently with the extensional pre-unification rules.

Primitive substitution is needed in higher-order resolution theorem proving for completeness reasons. We illustrate the importance of primitive substitution with the help of our example clauses \( \mathcal{C}_{11}, \mathcal{C}_{25}, \) and \( \mathcal{C}_{31} \) expressing that the relation represented by variable \( V^1 \) is reflexive, symmetric, and transitive. Remember that these clauses have been derived from (the negation of) our example problem (1)–(5). In order to prove theorem (5) we need to refute the clause set \( \{ \mathcal{C}_{11}, \mathcal{C}_{25}, \mathcal{C}_{31} \} \). Without guessing a candidate relation for \( V^1 \), Leo-II cannot find a refutation. Guessing instantiations of free relation variables, such as \( V^1 \), occurring at head positions in literals is the task of the primitive substitution rule. In fact, Leo-II proposes the instance \( \lambda X_0 \lambda Y_0 \bot \), see (7) above. Then it quickly finds a contradiction to reflexivity; see the proof protocol excerpt in Fig. 3.
Simplification, Rewriting, Subsumption, Heuristic Control, etc. Simplification, rewriting, and subsumption are still at comparably early stage of development. However, we believe that with the help of our term indexing mechanism (see Sec. 6) we shall be able to develop efficient solutions soon. Intelligent heuristic control, for example, based on term orderings, is also future work.

Leo-II is incomplete (and probably always will be) As has been pointed out already, Leo-II sacrifices completeness in various respects for pragmatic reasons. When our current prototype development has sufficiently progressed, our first interest will be to adapt Leo-II to particular problem classes and to make it strong for them. We have a particular interest in optimizing the reasoning in Leo-II towards quick transformation of essentially higher-order clauses into essentially first-order or propositional ones as illustrated before (see $D_1 - D_4$). Then, as we will further discuss in Sec. 4, we want to cooperate with specialist reasoners for efficient refutation of these subsets of clauses. In summary, we are rather interested in the strengths of the cooperative approach than in isolated completeness of Leo-II and both aspects may even turn out to be in conflict with each other.

4 Cooperation with Specialist Reasoners

Leo-II tries to operate on essentially higher-order clauses with the goal to reduce them to sets of essentially first-order or essentially propositional ones. Ideally,
the latter sets grow until they eventually become efficiently refutable by an automated specialist reasoner. Fig. 4 graphically illustrates this idea.

We also illustrate the idea by a slight modification of our running example. Instead of proving (5) from (1)–(4) we now want to prove (8) from (1)–(4):

\[ \text{equiv}_{\text{rel}} = \] (8)

Compared to (5) this is now an even simpler problem and it is clearly not higher-order anymore. In fact, definition unfolding and clause normalization immediately turns this problem into a trivially refutable set of equations containing not a single free variable, see the clauses 26, 28, 38, 39, 40, and 41 in Fig. 3. In the general case, however, the set of essentially first-order clauses and essentially propositional clauses generated by Leo-II this way may easily become very large. Since Leo-II’s proof search is not tailored for efficient first-order or propositional reasoning it may therefore fail to prove them. Therefore, a main objective of Leo-II is to fruitfully cooperate with first-order reasoners.

In Fig. 5 we illustrate the cooperative proof search approach by showing an excerpt of a Leo-II session in which the generated essentially first-order clauses 26, 28, 38, 39, 40, and 41 are passed to prover E [25] for refutation.

Here are some general remarks on our approach:

– The idea is to cooperate with different specialist reasoners for fragments of higher-order logic. Currently we are experimenting with the first-order systems E and SPASS. Nevertheless our idea is generic and Leo-II may later support also other fragments, with propositional logic and monadic second-order logic as possible candidates.
– As has been shown by Hurd [16] and Meng/Paulson [20] the particular syntax translations used to convert essentially higher-order clauses to essentially first-order ones may have a strong impact on the efficiency of the specialist provers. Thus, we later want to support a wide range of such syntax transformations. Currently our transformation approach is based on Kerber’s PhD thesis [17]. For communication we use the TPTP FOF syntax [27].
– The specialist reasoners will run in parallel to Leo-II and they will incrementally receive new clauses from Leo-II belonging to their fragment. Once they find a refutation, they report this back to Leo-II, which then reports that a proof has been found. However, this has not been fully realized yet.
– We are interested in producing proof objects that are verifiable by the proof assistant Leo-II is working for, in the first place, Isabelle/HOL. Leo-II already produces detailed proof objects in (extended) TSTP format. The results of the specialist reasoners, however, can currently not be translated back into Leo-II (sub-)proofs. This is future work.

5 Leo-II as an Interactive Proof Assistant

Leo-II provides an interactive mode in which user and system can interact to produce proofs in simple type theory. So far Leo-II offers 49 commands, roughly
1. (equiv_rel @ (^ \[X:$.i,Y:$.i\] : (X = Y))) = $true
   --- theorem(file('..//problems/SIMPLE-MATHS-3.thf',[thm]))
2. (equiv_rel @ (^ \[X:$.i,Y:$.i\] : (X = Y))) = $false
   --- neg_input 1
3. (~ ((~ (! \[x0:$.i,x1:$.i\] : (x0 = x0))) | (~ (~ ((~ (! \[x0:$.i,x1:$.i\] : ((~ (x0 = x1)) | (x1 = x0)))) | (~ (! \[x0:$.i,x1:$.i,x2:$.i\] : ((~ (x0 = x1))) | (x1 = x2))))))) = $false
   --- unfold_def 2
26. (sk_x0_1 = sk_x0_1) = $false | (sk_x1_4 = sk_x0_3) = $false | (sk_x0_5 = sk_x2_9) = $false
   --- cnf 24
28. (sk_x0_1 = sk_x0_1) = $false | (sk_x0_3 = sk_x1_4) = $true | (sk_x0_6 = sk_x2_10) = $false
   --- cnf 26
28. (sk_x0_4 = sk_x0_3) = $false | (sk_x0_1 = sk_x0_1) = $false | (sk_x0_5 = sk_x1_7) = $true
   --- cnf 35
39. (sk_x1_6 = sk_x0_3) = $false | (sk_x0_1 = sk_x0_1) = $false | (sk_x1_7 = sk_x2_9) = $true
   --- cnf 34
40. (sk_x0_3 = sk_x0_1) = $true | (sk_x0_1 = sk_x0_1) = $false | (sk_x0_6 = sk_x1_8) = $true
   --- cnf 37
41. (sk_x1_4 = sk_x0_1) = $true | (sk_x0_1 = sk_x0_1) = $false | (sk_x1_8 = sk_x2_10) = $true
   --- cnf 36

LED-II> call-fo-atp e
*** File /tmp/atp_in written; it contains translations of the FO-like clauses in LED-II's search space into FOPFPTP FOF syntax. Here is its content: ***
fof(leo_II_clause_26,axiom,((~ (sk_x1_4 = sk_x0_3)) | ((~ (sk_x0_6 = sk_x2_9)) | (~ (sk_x0_1 = sk_x0_1)))).
fof(leo_II_clause_28,axiom,((~ (sk_x0_4 = sk_x0_3)) | ((sk_x0_3 = sk_x1_4) | ((sk_x0_6 = sk_x2_10)) | (sk_x0_1 = sk_x0_1)))).
fof(leo_II_clause_38,axiom,((~ (sk_x1_4 = sk_x0_3)) | ((sk_x0_5 = sk_x1_7) | ((~ (sk_x0_1 = sk_x0_1)) | ((sk_x0_6 = sk_x2_9)) | (~ (sk_x1_4 = sk_x0_3)) | (~ (sk_x1_7 = sk_x2_9)) | (~ (sk_x0_1 = sk_x0_1))))).
fof(leo_II_clause_39,axiom,((~ (sk_x1_4 = sk_x0_3)) | ((sk_x0_5 = sk_x1_7) | ((sk_x1_6 = sk_x0_3)) | (~ (sk_x1_4 = sk_x0_3)) | (~ (sk_x1_7 = sk_x2_9)) | (~ (sk_x0_1 = sk_x0_1)))).
fof(leo_II_clause_40,axiom,((~ (sk_x0_6 = sk_x1_8)) | ((sk_x0_3 = sk_x1_4) | (~ (sk_x0_1 = sk_x0_1)))).
fof(leo_II_clause_41,axiom,((~ (sk_x1_8 = sk_x2_10)) | ((sk_x0_3 = sk_x1_4) | (~ (sk_x0_1 = sk_x0_1))))).
*** End of file /tmp/atp_in ***
*** Calling the first order ATP E on /tmp/atp_in ***
*** Result of calling first order ATP E on /tmp/atp_in ***
# Proof found!
# SZS status: Unatisfiable
# Initial clauses: : 6
# Removed in preprocessing: : 0
# Initial clauses in saturation: : 6
# Processed clauses: : 11
# ...of these trivial: : 0
# ...subsumed: : 0
# ...remaining for further processing: : 11
# Current number of unprocessed clauses: 0
# ...number of literals in the above: : 0
# Clause-clause subsumption calls (MU): 1
# Rec. Clause-clause subsumption calls: 1
# Unit Clause-clause subsumption calls: 0
# Rewrite failures with RHS unbound: : 0
*** End of file /tmp/atp_out ***
LED-II>

Fig. 5. For problem (8) we immediately generate a set of essentially first-order clauses which are here refuted by the first-order prover E.
half of which support interactive proof construction. The others are of general nature and support tasks like reading a problem from a file or inspection of Leo-II’s search state.

As illustrated before, the user may also interactively call external specialist reasoners from Leo-II. Thus, in interactive mode, Leo-II is a lean but nevertheless fully equipped proof assistant for simple type theory. The main difference to systems such as HOL4, Isabelle/HOL, and OMEGA is that Leo-II’s base calculus is extensional higher-order resolution, which is admittedly not very well suited for developing complex proofs interactively. However, for training students in higher-order resolution based theorem proving, for debugging, and for presentation purposes our interactive mode may turn out to be very useful.

Leo-II in particular provides support to investigate the proof state, the internal data representation (shared terms), and term index. For example, the command `termgraph-to-dot` generates a graphical representation of Leo-II’s shared term data structure, which is a directed acyclic graph, in the DOT syntax [13] which can be processed by the program `dot` [13] in order obtain, for example, the ps-representation as given in Fig. 6.

With the command `analyze-index` we may request useful statistical information about our term data structure, such as the term sharing rate. We may, for instance, analyze how prove search modifies the term graph and changes the term sharing rate when applied to our running example from Fig. 2. For this, we call `analyze-index` first after loading the problem and then again after successfully proving it:

```
LEO-II> read-problem-file ../problems/SIMPLE-MATHS-5.thf
LEO-II> analyze-index
[...]
------------- The Termset Analysis -------------
[...]
Sharing rate: 8 nodes with 7 bindings
Average sharing rate: 0.878 bindings per node
Average term size: 2.75
Average number of supernodes: 2.25
Average number of supernodes (symbols): 2.66666666667
Average number of supernodes (nonprimitive terms): 1.5
Rate of term occurrences PST size / term size: 0.440298507463
Rate of symbol occurrences PST size / term size: 0.510204081633
Rate of bound occurrences PST size / term size: 0.636366364
------------- End Termset Analysis -------------
LEO-II> prove
[...]
Eureka --- Thanks to Corina!
[...]
LEO-II (Proof Found!)> analyze-index
[...]
------------- The Termset Analysis -------------
[...]
Sharing rate: 232 nodes with 325 bindings
Average sharing rate: 1.40086206897 bindings per node
Average term size: 11.0689655172
Average number of supernodes: 7.62293103448
Average number of supernodes (symbols): 10.6060606061
Average number of supernodes (nonprimitive terms): 5.7150376344
Rate of term occurrences PST size / term size: 0.228156769104
Rate of symbol occurrences PST size / term size: 0.38625666713
Rate of bound occurrences PST size / term size: 0.504699009398
------------- End Termset Analysis -------------
```
Operations on terms in Leo-II are supported by term indexing. Key features of Leo-II’s term indexing are the representation of terms in a perfectly shared graph structure and the indexing of various structural properties, such as the occurrence of subterms and their position.

Term sharing is widely employed in first-order theorem proving [24, 25, 28]: syntactically equal terms are represented by a single instance. For Leo-II, we have adapted this technique to the higher-order case. We use de Bruijn-notation [10] to avoid blurring of syntactical equality by α-conversion.

A shared representation of terms has multiple benefits. The most obvious is the instant identification of all occurrences of a term or subterm structure. Furthermore, it allows an equality test of syntactic structures at constant cost, which allows the pruning of structural recursion over terms early in many operations. Finally, it allows for ‘tabling of term properties’ (i.e., the memorization of term properties with the help of tables) at reasonable cost, as the extra effort spent on term analysis is compensated by the reusability of the results.

The indexing approach of Leo-II, which is employed, for example, to determine candidate clauses for extensional resolution and subsumption, has a strong focus on structural aspects. It differs in this respect from the approach by Pientka [23], which is based on a discrimination tree and which allows for perfect filtering on the basis of higher order pattern unification. In contrast, we are particularly interested also in more relaxed search criteria, such as subterm occurrences or head symbols.

Equality and Occurrences The basis of Leo-II’s data structure for terms is the shared representation of all occurrences of a syntactical structure by exactly one instance. This invariant is preserved by all operations on the index, such as insertion of new terms to the index. An example of a shared term representation, called term graph, is shown in Fig. 6.

Leo-II’s term graph is implemented in a data structure based on hashtables. Based on the invariant of a perfectly shared representation of terms in the graph, the check for the existence of a given syntactical structure in the index is reduced to a few hashtable lookups. Our representation in particular reduces equality checks for terms in the index to a single pointer comparison. In addition, the index provides information on the structure of terms by indexing subterm occurrences. From Leo-II’s interactive interface, such information can be accessed by the user using the command inspect-node and inspect-symbol.

We exemplarily apply the command inspect-symbol after automatically proving our running example to the variable symbol $V_x0_1$:

```
LEO-II> read-problem-file ../problems/SIMPLE-MATHS-5.thf
LEO-II> prove
[...]
Eureka --- Thanks to Corina!
[...]
LEO-II (Proof Found!)> inspect-symbol V_x0_1
```
Inspecting:
node 161: V_x0_1
Type:
$\emptyset > ($i > $o)
Structure:
symbol V_x0_1
Parents:
- as function term:
  node 180: V_x0_1 @ (sk_x2_4 @ V_x0_1)
  node 174: V_x0_1 @ (sk_x1_6 @ V_x0_1)
  node 164: V_x0_1 @ (sk_x1_1 @ V_x0_1)
  node 178: V_x0_1 @ (sk_x2_7 @ V_x0_1)
  node 168: V_x0_1 @ (sk_x1_3 @ V_x0_1)
- as argument term:
  node 182: sk_x3_5 @ V_x0_1
  node 173: sk_x1_6 @ V_x0_1
  node 163: sk_x1_1 @ V_x0_1
  node 176: sk_x2_7 @ V_x0_1
  node 167: sk_x1_3 @ V_x0_1
  node 170: sk_x2_4 @ V_x0_1

Total: 11 parents

Occurs in terms indexed with role:
node 165: (V_x0_1 @ (sk_x1_1 @ V_x0_1)) @ (sk_x1_1 @ V_x0_1)
  (in Clause:25/0 max neg)
node 171: (V_x0_1 @ (sk_x1_3 @ V_x0_1)) @ (sk_x2_4 @ V_x0_1)
  (in Clause:33/1 max pos)
node 177: (V_x0_1 @ (sk_x1_6 @ V_x0_1)) @ (sk_x2_7 @ V_x0_1)
  (in Clause:28/2 max pos)
node 179: (V_x0_1 @ (sk_x2_7 @ V_x0_1)) @ (sk_x1_6 @ V_x0_1)
  (in Clause:25/2 max neg)
node 183: (V_x0_1 @ (sk_x2_4 @ V_x0_1)) @ (sk_x3_5 @ V_x0_1)
  (in Clause:35/1 max pos)
node 184: (V_x0_1 @ (sk_x3_3 @ V_x0_1)) @ (sk_x3_5 @ V_x0_1)
  (in Clause:28/4 max neg)

Total: 6 terms

LEO-II (Proof Found!)

The index first provides information on the direct relation of $V_{x0_1}$ to other terms in the index: $V_{x0_1}$ occurs as function term or argument in the terms represented by the 11 parent nodes shown above. This information is maintained in cascaded hash tables and provides the necessary information to preserve the index’ single instance representation. For example, when a term $sk_{x1_6} @ V_{x0_1}$ is inserted to the index in the displayed state, the existence of a node representing this term can be checked by two hashtable lookups: first all terms that have variable $V_{x0_1}$ as argument are looked up, then a second hashtable lookup tests whether there is among these a term with function term $sk_{x1_6}$. Here, in fact, we already have a node in the index representing this term, namely node 173.

The single instance representation allows for indexing of term properties, such as the occurrence in clause literals or in other relevant positions: In the given proof state, the variable $V_{x0_1}$ occurs in 6 clause literals, where three of them are literals of clause 25:

25: (($V_{x0_1} @ (sk_{x1_1} @ V_{x0_1})) @ (sk_{x1_1} @ V_{x0_1})) = false |
    (($V_{x0_1} @ (sk_{x1_3} @ V_{x0_1})) @ (sk_{x3_5} @ V_{x0_1})) = false |
    (($V_{x0_1} @ (sk_{x3_5} @ V_{x0_1})) @ (sk_{x1_6} @ V_{x0_1})) = false

A term graph representing these three literals is shown in Fig. 6. Node 161, representing variable $V_{x0_1}$, is shared by the nodes 165, 184 and 179, as it is reachable from all of them. Furthermore, all occurrences within a single term
are also represented by a single node. The graph shows node 161 occurring in six different positions in the three literals, both as a function term of some application (shown by filled arrowheads) and as argument term (shown by blank arrowheads). However, not only primitive terms such as symbols are shared, but also non-primitive terms, that is, applications and abstractions. An example is node 163 \((\text{sk}_x^1 \vartriangleleft V_x^0 \text{.1})\), which is shared by nodes 164 \((V_x^0 \text{.1} \vartriangleleft (\text{sk}_x^1 \vartriangleleft V_x^0 \text{.1}))\) and 165 \(((V_x^0 \text{.1} \vartriangleleft (\text{sk}_x^1 \vartriangleleft V_x^0 \text{.1}))) \vartriangleleft (\text{sk}_x^1 \vartriangleleft V_x^0 \text{.1}))\).

Using the index In Leo-II, the information provided by the index is used to guide a number of operations both at term level as well as at calculus level. In addition to speeding up standard operations in the proving procedure, the indexing mechanism allows us to address problems in a different way. For example, it helps avoiding a naive, sequential checking of many clause and literal properties. Instead, the checking process is reversed and terms in the index having a particular property are first identified and then the relevant clauses are selected via hashtable lookups. Global unfolding of definition is already in Leo-II this way. Unfortunately this is not the case yet for many other important components, including simplification, rewriting, and subsumption.

7 Related Work

The integration of reasoners and reasoning strategies was pioneered in the TEAMWORK system [12], which realizes the cooperation of different reasoning strategies, and the TECHS system [11], which realizes a cooperation between a set of heterogeneous first-order theorem provers. Related is also the work of Meier [18], Hurd [16], and Meng/Paulson [19, 21]. They realize interfaces between proof assistants (OMEGA, HOL, and Isabelle/HOL) and first-order theorem provers.
All these approaches pass essentially first-order clauses to first-order theorem provers after appropriate syntax transformations. The main difference to the work presented here is that Leo-II calls first-order provers from within automatic higher-order proof search.

The project Leo-II was strongly inspired by encouraging previous work on LEO and the agent based OANTS framework [3, 2, 4].

8 Concluding Remarks

The Leo-II project has been under way since October 2006. As this document illustrates the Leo-II system has since developed comparably fast with respect to both its interactive and its automatic mode. Within short time we have produced 11662 lines of OCAML code (partly as OCAML beginners). While still facing several ‘Kinderkrankheiten’ we have now entered the highly fascinating theorem prover development phase in which first theorems can already be proven automatically although some crucial features, in particular wrt. heuristic guidance layer, are still missing or have to be further investigated and developed.

As the system is still changing rapidly an extensive case study at this stage would be quite meaningless. However, such case studies are planned for the final phase of the project.

References


Abstract We present an extensible encoding of object-oriented data models into higher-order logic (HOL). Our encoding is supported by a datatype package that enables the use of the shallow embedding technique to object-oriented specification and programming languages. The package incrementally compiles an object-oriented data model, i.e., a class system, to a theory containing object-universes, constructors, and accessor functions, coercions (casts) between dynamic and static types, characteristic sets, their relations reflecting inheritance, and co-inductive class invariants. The package is conservative, i.e., all properties are derived entirely from constant definitions. As an application, we show constraints over object structures.

1 Introduction

While object-oriented (OO) programming is a widely accepted programming paradigm, theorem proving over object-oriented programs or object-oriented specifications is far from being a mature technology. Classes, inheritance, subtyping, objects and references are deeply intertwined and complex concepts that are quite remote from the platonic world of first-order logic or higher-order logic (HOL). For this reason, there is a tangible conceptual gap between the verification of functional programs on the one hand and object-oriented programs on the other. This is mirrored in the increasing limitations of proof environments.

The existing proof environments dealing with subtyping and references can be categorized as: 1) verification condition generators reducing a Hoare-style proof into a proof in a standard logic, and 2) deep embeddings into a meta-logic. Verification condition generators, for example, are Boogie for Spec# [2,12], Krakatoa [13] and several similar tools based on the Java Modeling Language (JML). The underlying idea is to compile object-oriented programs into standard imperative ones and to apply a verification condition generator on the latter. While technically sometimes very advanced, the foundation of these tools is quite problematic: The generators usually supporting a large language fragment are not verified, and it is not clear if the generated conditions are sound and complete with respect to the (usually complex) operational semantics.

Among the tools based on deep embeddings, there is a sizable body of literature on formal models of Java-like languages (e.g., [9,10,19,23]). In a deep
embedding of a language semantics, syntax and types are represented by free datatypes. As a consequence, derived calculi inherit a heavy syntactic bias in form of side-conditions over binding and typing issues. This is unavoidable if one is interested in meta-theoretic properties such as type-safety; however, when reasoning over applications and not over language tweaks, this advantage turns into a major obstacle for efficient deduction. Thus, while various proofs for type-safety, soundness of Hoare calculi and even soundness of verification condition generators are done, none of the mentioned deep embeddings has been used for substantial proof work in applications.

In contrast, the shallow embedding technique has been used for semantic representations such as HOL itself (in Isabelle/Pure), for HOLCF (in Isabelle/HOL) allowing reasoning over Haskell-like programs [16] or for HOL-Z [6,3].

The essence of an effective shallow embedding is to find an injective mapping of the pair of an object language expression \( E \) and its type \( T \) to a pair \( E :: T \) of the meta-language HOL. “Injective mapping” means, that well-typedness is preserved in both ways. Thus, type-related side-conditions in derived object-language calculi can be left implicit. Since such implicit side-conditions are “implemented” by a built-in mechanism of the meta-logic, they can be checked orders of magnitude faster compared to an explicit treatment involving tactic proof.

At first sight, it seems impossible to apply the injective shallow embedding technique to object-oriented languages: Their characteristic features like subtyping and inheritance are not present in the typed \( \lambda \)-calculi underlying HOL systems. However, an injective mapping does mean a simple one-to-one conversion; rather, the translation might use a pre-processing making, for example, implicit casts between subtypes and supertypes explicit. Still, this requires a data model that respects semantic properties like no loss of information in casts.

Beyond the semantical requirements, there is an important technical one: object-oriented data models must be extensible, i.e., it must be possible to add to an existing class system a new class without reproving everything. The problem becomes apparent when considering the underlying state of an object-oriented program called object structure. Objects are abstract representations of pieces of memory that are linked via references (object identifiers, oid) to each other. Objects are tuples of “class attributes,” i.e., elementary values like Integers or Strings or references to other objects. The type of these tuples is viewed as the type of the class they are belonging to. Object structures (or: states) are maps of type \( \text{oid} \rightarrow U \) relating references to objects living in a universe \( U \) of all objects.

Instead of constructing such a universe globally for all data-models (which is either single-typed and therefore not an injective type representation, or “too large” for the type system of HOL), one could think of generating an object universe only for each given class system. Ignoring subtyping and inheritance for a moment, this would result in a universe \( U^0 = A + B \) for some class system with the classes \( A \) and \( B \). Unfortunately, such a construction is not extensible: If we add a new class to an existing class system, say \( D \), then the “obvious” construction \( U^1 = A + B + D \) results in a type different from \( U^0 \), making their object structures logically incomparable. Properties, that have been proven over
\(U^0\) will not hold over \(U^1\). Thus, such a naive approach rules out an incremental construction of class systems, which makes it unfeasible in practice.

As contributions of this paper, we present a novel universe construction which represents subtyping within parametric polymorphism in an injective, type-safe manner and which is extensible. This construction is implemented in a datatype-package for Isabelle/HOL, i.e., a kind of logic compiler that generates for each class system and its extensions conservative definitions. This includes the definition of constructors and accessors, casts between types, characteristic sets of objects. On this basis, properties reflecting subtyping and proof principles like class invariants are automatically derived.

2 Formal and Technical Background

Isabelle [18] is a generic, LCF-style theorem prover implemented in SML. For our object-oriented datatype package, we use the possibility to build SML programs performing symbolic computations over formulae in a logically safe way. Isabelle/HOL offers support for checks for conservatism of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableau provers.

Higher-order logic (HOL) [1] is a classical logic with equality enriched by total polymorphic higher-order functions. The type constructor for the function space is written infix: \(\alpha \Rightarrow \beta\); multiple applications like \(\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)\) are also written as \([\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}\). HOL is centered around the extensional logical equality \(_= _\) with type \([\alpha, \alpha] \Rightarrow \text{bool}\), where bool is the fundamental logical type.

We assume a type class \(\alpha :: \text{bot}\) for all types \(\alpha\) that provide an exceptional element \(\bot\); for each type in this class a test for defindness is available via \(\text{def x} \equiv (x \neq \bot)\). The HOL type constructor \(\bot\) assigns to each type \(\tau\) a type \(\tau\) lifted by \(\bot\). Thus, each type \(\alpha\) is member of the class bot. The function \(\iota : \alpha \Rightarrow \alpha\) denotes the injection, the function \(\iota : \alpha \Rightarrow \alpha\) its inverse for defined values. Partial functions \(\alpha \rightarrow \beta\) are just functions \(\alpha \Rightarrow \beta\).

3 Level 0: Typed Object Universes

In this section, we introduce our families \(\mathcal{U}^i\) of object universes. Each \(\mathcal{U}^i\) comprises all value types and an extensible class type representation induced by a class hierarchy. To each class, a class type is associated which represents the set of object instances or objects. The extensibility of a universe type is reflected by "holes" (polymorphic variables), that can be filled when "adding" extensions to a class. Our construction ensures that \(\mathcal{U}^{i+1}\) is just a type instance of \(\mathcal{U}^i\) (where \(\mathcal{U}^{i+1}\) is constructed by adding new classes to \(\mathcal{U}^i\)). Thus, properties proven over object systems “living” in \(\mathcal{U}^i\) remain valid in \(\mathcal{U}^{i+1}\).
3.1 A Formal Framework of Object Structure Encoding

We will present the framework of our object encoding together with a small example: assume a class \texttt{Node} with an attribute \texttt{i} of type integer and two attributes \texttt{left} and \texttt{right} of type \texttt{Node}, and a derived class \texttt{Cnode} (thus, \texttt{Cnode} is a subtype of \texttt{Node}) with an attribute \texttt{color} of type \texttt{Boolean}.

In the following we define several type sets which all are subsets of the types of the HOL type system. This set, although denoted in usual set-notation, is a meta-theoretic construct, i.e., it cannot be formalized in HOL.

**Definition 1 (Attribute Types).** The set of attribute types \( \mathcal{A} \) is defined inductively as follows:
1. \( \{ \text{Boolean, Integer, Real, String, oid} \} \subset \mathcal{A} \), and
2. \( \{ a \text{ Set, a Sequence, a Bag} \} \subset \mathcal{A} \) for all \( a \in \mathcal{A} \).

Attributes with class types, e.g., the attribute \texttt{left} of class \texttt{Node}, are encoded using the abstract type \texttt{oid}. These object identifiers (i.e., references) will be resolved by accessor functions like \( \texttt{A.left(1)} \) for a given state; an access failure will be reported by \( \bot \).

In principle, a class is a Cartesian products of its attribute types extended by an abstract type ensuring uniqueness.

**Definition 2 (Tag Types).** For each class \( C \) a tag type \( t \in \mathcal{T} \) is associated. The set \( \mathcal{T} \) is called the set of tag types.

Tag types are one of the reasons why we can build a strongly typed universe (with regard to the object-oriented type system), e.g., for class \texttt{Node} we assign an abstract datatype \texttt{Node\_t} with the only element \texttt{Node\_key}. Further, for each class we introduce a base class type:

**Definition 3 (Base Class Types).** The set of base class types \( \mathcal{B} \) is defined as follows:
1. classes without attributes are represented by \( (t \times \text{unit}) \in \mathcal{B} \), where \( t \in \mathcal{T} \) and \text{unit} is a special HOL type denoting the empty product.
2. if \( t \in \mathcal{T} \) is a tag type and \( a_i \in \mathcal{A} \) for \( i \in \{0, \ldots, n\} \) then \( (t \times a_0 \times \cdots \times a_n) \in \mathcal{B} \).

Thus, the base object type of class \texttt{Node} is \texttt{Node\_t \times Integer \times oid \times oid} and of class \texttt{Cnode} is \texttt{Cnode\_t \times Boolean}.

Without loss of generality, we assume in our object model a common supertype of all objects. For example, for OCL (Object Constraint Language), this is \texttt{OclAny}, for Java this is \texttt{Object}.

**Definition 4 (Object).** Let \( t \in \mathcal{T} \) be the tag of the common supertype \texttt{Object} and \texttt{oid} the type of the object identifiers we define \( \alpha \texttt{Object} := ((Object \times oid) \times \alpha_\bot) \).

Object generator functions can be defined such that freshly generated object-identifiers to an object are also stored in the object itself; thus, the construction of reference types and of referential equality is fairly easy (see the discussion on state invariants in Section 7.4). Now we have all the foundations for defining the type of our family of universes formally:
Definition 5 (Universe Types). The set of all universe types $\mathcal{U}_{ref}$ resp. $\mathcal{U}_{uref}$ (abbreviated $\mathcal{U}_r$) is inductively defined by:

1. $\mathcal{U}_r^0 \in \mathcal{U}_r$ is the initial universe type with one type variable (hole) $\alpha$.

2. If $\mathcal{U}_{(\alpha_0, \ldots, \alpha_n, \beta_1, \ldots, \beta_m)} \in \mathcal{U}_r$, $n, m \in \mathbb{N}$, $i \in \{0, \ldots, n\}$ and $c \in \mathcal{B}$ then
   \[ \mathcal{U}_{(\alpha_0, \ldots, \alpha_i, \beta_1, \ldots, \beta_m)} \times ((c \times (\alpha_{n+1})_\perp + \beta_{m+1})) \in \mathcal{U}_r \]

This definition covers the introduction of “direct object extensions” by instantiating $\alpha$-variables.

3. If $\mathcal{U}_{(\alpha_0, \ldots, \alpha_n, \beta_1, \ldots, \beta_m)} \in \mathcal{U}_r$, $n, m \in \mathbb{N}$, $i \in \{0, \ldots, m\}$, and $c \in \mathcal{B}$ then
   \[ \mathcal{U}_{(\alpha_0, \ldots, \alpha_i, \beta_1, \ldots, \beta_m)} \times ((c \times (\alpha_{n+1})_\perp + \beta_{m+1})) \in \mathcal{U}_r \]

This definition covers the introduction of “alternative object extensions” by instantiating $\beta$-variables.

The initial universe $\mathcal{U}_r^0$ represents mainly the common supertype (i.e., Object) of all classes, i.e., a simple definition would be $\mathcal{U}_r^0 = \alpha \text{Object}$. However, we will need the ability to store Values $= \text{Real} + \text{Integer} + \text{Boolean} + \text{String}$. Therefore, we define the initial universe type by $\mathcal{U}_r^0 = \alpha \text{Object} + \text{Values}$. Extending the initial universe $\mathcal{U}_r^0$, in parallel, with the classes $\text{Node}$ and $\text{Cnode}$ leads to the following universe type:

\[ \mathcal{U}_{(\alpha, \beta_c, \beta_n)} = (\text{Node}_1 \times \text{Integer} \times \text{oid} \times \text{oid}) \times ((\text{Cnode}_e \times \text{Boolean} \times (\alpha_\perp + \beta_c) \perp + \beta_n)) \text{Object} + \text{Values}. \]

We pick up the idea of a universe representation without values for a class with all its extensions (subtypes). We construct for each class a type that describes a class and all its subtypes. They can be seen as “paths” in the tree-like structure of universe types, collecting all attributes in Cartesian products and pruning the type sums and $\beta$-alternatives.

Definition 6 (Class Type). The set of class types $\mathcal{C}$ is defined as follows: Let $\mathcal{U}$ be the universe covering, among others, class $C_n$, and let $C_0, \ldots, C_{n-1}$ be the supertypes of $C$, i.e, $C_i$ is inherited from $C_{i-1}$. The class type of $C$ is defined as:

1. $C_i \in \mathcal{U}_r, i \in \{0, \ldots, n\}$ then
   \[ \mathcal{C}^0 = (C_0 \times (C_1 \times (C_2 \times \ldots \times (C_n \times \alpha_\perp) \perp) \perp)_\perp) \in \mathcal{C}, \]

2. $\mathcal{U}_c \supseteq \mathcal{C}$, where $\mathcal{U}_c$ is the set of universe types with $\mathcal{U}_r^0 = \mathcal{C}^0$.

Thus in our example we construct for the class type of class $\text{Node}$ the type

\[ (\alpha_C, \beta_C) \text{Node} = (\text{Node}_1 \times \text{Integer} \times \text{oid} \times \text{oid}) \times ((\text{Cnode}_e \times \text{Boolean} \times (\alpha_C \perp + \beta_C) \perp) \text{Object}). \]

Here, $\alpha_C$ allows for extension with new classes by inheriting from $\text{Cnode}$ while $\beta_C$ allows for direct inheritance from $\text{Node}$. 

The outermost \( \bot \) reflect the fact that class objects may also be undefined, in particular after projecting them from some term in the universe or failing type casts. Thus, also the arguments of constructor may be undefined.

### 3.2 Handling Instances

We provide for each class injections and projects. In the case of \( \text{Object} \) these definitions are quite easy, e.g., using the constructors \( \text{Inl} \) and \( \text{Inr} \) for type sums we can easily insert an \( \text{Object} \) object into the initial universe via

\[
\text{mk}_{\text{Object}}^{(0)} \text{self} = \text{Inl self} \quad \text{with type } \alpha \text{Object} \Rightarrow V_\alpha^0
\]

and the inverse function for constructing an \( \text{Object} \) object out of an universe can be defined as follows:

\[
\text{get}_{\text{Object}}^{(0)} u = \begin{cases} 
  k & \text{if } u = \text{Inl } k \\
  \varepsilon.k.\text{true} & \text{if } u = \text{Inr } k
\end{cases} \quad \text{with type } V_\alpha^0 \Rightarrow \alpha \text{Object}.
\]

In the general case, the definitions of the injections and projections is a little bit more complex, but follows the same schema: for the injections we have to find the “right” position in the type sum and insert the given object into that position. Further, we define in a similar way projectors for all class attributes. For example, we define the projector for accessing the \( \text{left} \) attribute of the class \( \text{Node} \):

\[
\text{self}.\text{left}^{(0)} \equiv (\text{fst} \circ \text{snd} \circ \text{snd} \circ \text{fst}) \circ \text{base } \text{self}
\]

with type \((\alpha_1, \beta) \text{Node} \Rightarrow \text{oid}\) and where base is a variant of \(\text{snd}\) over lifted tuples:

\[
\text{base } x \equiv \begin{cases} 
  b & \text{if } x = (a, b) \\
  \bot & \text{else}
\end{cases}
\]

For attributes with object types we return an \(\text{oid}\). In Section 5, we show how these projectors can be used for defining a type-safe variant.

In a next step, we define type test functions; for universe types we need to test if an element of the universe belongs to a specific type, i.e., we need to test which corresponding extensions are defined. For \(\text{Object}\) we define:

\[
\text{isUniv}_{\text{Object}}^{(0)} \quad u = \begin{cases} 
  \text{true} & \text{if } u = \text{Inl } k \\
  \text{false} & \text{if } u = \text{Inr } k
\end{cases} \quad \text{with type } V_\alpha^0 \Rightarrow \text{bool}.
\]

For class types we define two type tests, an exact one that tests if an object is exactly of the given \textit{dynamic type} and a more liberal one that tests if an object is of the given type or a subtype thereof. Testing the latter one, which is called
kind in the OCL standard, is quite easy. We only have to test that the base type of the object is defined, e.g., not equal to ⊥:

$$\text{isKind}(0)_{\text{Object}} \circ \text{self} = \text{def} \text{self} \quad \text{with type } \alpha \text{Object} \Rightarrow \text{bool.}$$

An object is exactly of a specific dynamic type, if it is of the given kind and the extension is undefined, e.g.:

$$\text{isType}(0)_{\text{Object}} \circ \text{self} = \text{isKind}(0)_{\text{Object}} \land \neg(\text{def} \circ \text{base}) \text{self} \quad \text{of type } \alpha \text{Object} \Rightarrow \text{bool.}$$

The type tests for user defined classes are defined in a similar way by testing the corresponding extensions for definedness.

Finally, we define casts, i.e., ways to convert classes along their subtype hierarchy. Thus we define for each class a cast to its direct subtype and to its direct supertype. We need no conversion on the universe types where the subtype relations are handled by polymorphism. Therefore we can define the type casts as simple compositions of projections and injections, e.g.:

$$\text{Node}(0)_{\text{Object}} = \text{get}(0)_{\text{Object}} \circ \text{mk}(0)_{\text{Node}} \quad \text{of type } (\alpha_1, \beta_1) \text{Node} \Rightarrow (\alpha_1, \beta_1) \text{Object},$$

$$\text{Object}(0)_{\text{Node}} = \text{get}(0)_{\text{Node}} \circ \text{mk}(0)_{\text{Object}} \quad \text{of type } (\alpha_1, \beta_1) \text{Object} \Rightarrow (\alpha_1, \beta_1) \text{Node}.$$  

These type-casts are changing the static type of an object, while the dynamic type remains unchanged.

Note, for a universe construction without values, e.g., $\mathcal{U}^{0}_\alpha = \alpha \text{Object}$, the universe type and the class type for the common supertype are the same. In that case there is a particularly strong relation between class types and universe types on the one hand and on the other there is a strong relation between the conversion functions and the injections and projections function. In more detail, one can understand the projections as a cast from the universe type to the given class type and the injections are inverse.

Now, if we build a theorem over class invariants (based finally on these projections, injections, casts, characteristic sets, etc.), it will remain valid even if we extend the universe via $\alpha$ and $\beta$ instantiations. Therefore, we have solved the problem of structured extensibility for object-oriented languages.

This constructions establishes a subtype relation via inheritance. Therefore, a set of Nodes (with type $((\alpha_1, \beta) \text{Node}) \text{Set}$) can also contain objects of type $\text{Cnode}$. For resolving operation overloading, i.e., late-binding, the packages generates operation tables user-defined operations; see [7,5] for details.

## 4 Properties of the Object Encoding

Based on the presented definitions, our object-oriented datatype package proves that our encoding of object-structures is a faithful representation of object-oriented (e.g., in the sense of language like Java or Smalltalk or the UML standard [20]). These theorems are proven for each model, e.g., during loading a
specific class system. This is similar to other datatype packages in interactive theorem provers. Further, these theorems are also a prerequisite for successful reasoning over object structures.

In the following, we assume a model with the classes $A$ and $B$ where $B$ is a subclass of $A$. We start by proving this subtype relation for both our class type and universe type representation:

\[
\begin{align*}
\text{isUniv}^{(0)}_A u &
\quad \text{isType}^{(0)}_B \text{self} \\
\text{isUniv}^{(0)}_B u &
\quad \text{isKind}^{(0)}_A \text{self}
\end{align*}
\]

We also show that our conversion between universe representations and object representation is satisfy the no loss of information in casts-principle and that both type systems are isomorphic:

\[
\begin{align*}
\text{isUniv}^{(0)}_A u &
\quad \text{mk}^{(0)}_A (\text{get}^{(0)}_A u) = u \\
\text{isType}^{(0)}_A \text{self} &
\quad \text{get}^{(0)}_A (\text{mk}^{(0)}_A \text{self}) = \text{self}
\end{align*}
\]

\[
\begin{align*}
\text{isType}^{(0)}_B \text{self} &
\quad \text{isUniv}^{(0)}_A u \\
\text{isUniv}^{(0)}_A (\text{mk}^{(0)}_A \text{self}) &
\quad \text{isType}^{(0)}_A (\text{get}^{(0)}_A u)
\end{align*}
\]

Moreover, that we can “re-cast” an objects safely, i.e., the dynamic (class) type of an object can be casted to a supertype and back:

\[
\frac{\text{isType}^{(0)}_B \text{self}}{\text{isType}^{(0)}_B ((\text{get}^{(0)}_A \text{self})^{(0)}_A)^{(0)}_A)}
\]

The datatype package also shows similar properties for the injections and projections into attributes.

5 Level 1: A Type-safe Object Store

based on the concept of object universes, we define the store as a partial map:

\[
\alpha \text{St} := \text{oid} \to \mathcal{U}_\alpha.
\]

Since all operations over our object store will be parametrized by $\alpha \text{St}$, we introduced the following type synonym:

\[
V_\alpha(\tau) := \alpha \text{St} \Rightarrow \tau.
\]

Thus we can define type-safe accessor functions, i.e., object identifiers (references) are completely encapsulated:

\[
\text{self}.\text{left}(1) \sigma \equiv \begin{cases} \text{get}^{(0)}_{\text{Node}} u & \text{if } \sigma((\text{self} \sigma).\text{left}(0)) = \text{Some } u \\ \bot & \text{else} \end{cases}
\]
The object language type \texttt{.left : Node -> Node} is now represented by our construction with type $$\nu_{(\alpha C, \beta C)}(\nu_{(\alpha C, \beta C)}(\nu_{(\alpha C, \beta C)}(\text{Node}))) \Rightarrow \nu_{(\alpha C, \beta C)}(\nu_{(\alpha C, \beta C)}(\text{Node}))$$.

Thus, the representation map is injective on types; subtyping is represented by type-instantiation on the HOL-level. However, due to our universe construction, the theory on accessors, casts, etc. is also extensible.

All other operations like casting, type- or kind-check are lifted analogously; here we present only the case of the cast:

\[
\text{self}^{(1)}_{\sigma} \equiv (\text{self}^{(0)}_{\sigma})_{|A|}
\]

Moreover, the properties of the previous section were reformulated for this level.

With accessors and cast operations, we have now a path-language to access specific values in object structures. On top of this path language, we add a small assertion language to express properties: We write $$\sigma \models \partial x$$ for def$$(x \sigma)$$, and $$\sigma \not\models \partial x$$ for the contrary. With this predicate we can specify that the access to a value along a path succeeds in a given state.

Moreover, for arbitrary binary HOL operations \texttt{op} such as \texttt{=} \texttt{, _< }, \texttt{_<}, \texttt{_\&}, \texttt{_\rightarrow }, \ldots, \text{we write} \sigma \models P \texttt{ op } Q \text{ for } \llbracket P\sigma \rrbracket \texttt{ op } \llbracket Q\sigma \rrbracket$. Note that \texttt{_<} is underspecified for \texttt{\bot}, thus for illegal access into the state. An alternative semantic choice for the assertion language consists in a three-valued logic (cf. [7]).

6 Level 2: Co-inductive Properties in Object Structures

A main contribution of our work is the encoding of co-inductive properties object structures, including the support for class invariants.

Recall our previous example, where the class \texttt{Node} describes a potentially infinite recursive object structure. Assume that we want to constrain the attribute \texttt{i} of class \texttt{Node} to values greater than 5. This is expressed by the following function approximating the set of possible instances of the class \texttt{Node} and its subclasses:

\[
\text{NodeKindF} : \mathcal{U}_{(\alpha C, \beta C, \beta N)} \Rightarrow \mathcal{U}_{(\alpha C, \beta C, \beta N)} \Rightarrow (\alpha C, \beta C) \text{Node set}
\]

\[
\text{NodeKindF} \equiv \lambda \sigma. \lambda X. \{ \text{self} | \sigma \not\models \partial \text{self} \cdot \text{.i}^{(1)} \land \sigma \not\models \text{self} \cdot \text{.i}^{(1)} > 5 \\
\quad \land \sigma \not\models \partial \text{self} \cdot \text{.left}^{(1)} \land \sigma \not\models (\text{self} \cdot \text{.left}^{(1)}) \in X \\
\quad \land \sigma \not\models \partial \text{self} \cdot \text{.right}^{(1)} \land \sigma \not\models (\text{self} \cdot \text{.right}^{(1)}) \in X \}
\]

In a setting with subtyping, we need two characteristic type sets, a sloppy one, the characteristic kind set, and a fussy one, the characteristic type set. By adding the conjunct $$\sigma \models \text{isType}_{\text{Node}}^{(1)} \text{self}$$, we can construct another approximation function (which has obviously the same type as \text{NodeKindF}):

\[
\text{NodeTypeF} \equiv \lambda \sigma. \lambda X. \{ \text{self} | (\text{self} \in (\text{NodeKindF} \sigma X)) \\
\quad \land \sigma \not\models \text{isType}_{\text{Node}}^{(1)} \text{self} \}
\]
Thus, the characteristic kind set for the class Node can be defined as the greatest fixed-point over the function NodeKindF:

\[
\text{NodeKindSet} :: \mathbb{U}^1(\alpha_C, \beta_C, \beta_N) \Rightarrow \mathbb{U}^1(\alpha_C, \beta_C, \beta_N) \Rightarrow (\alpha_C, \beta_C)
\]

\[
\text{NodeKindSet} \equiv \lambda \sigma. (\text{gfp}(\text{NodeKindF } \sigma))
\]

For the characteristic type set we proceed analogously. We infer a class invariant theorem:

\[
\sigma \models \text{self} \in \text{NodeKindSet} = \sigma \models \partial \text{self}. i^{(1)} \land \sigma \models \text{self}. i^{(1)} > 5
\]

\[
\land \sigma \models \partial \text{self}. \text{left}^{(1)} \land \sigma \models (\text{self}. \text{left}^{(1)}) \in \text{NodeKindSet}
\]

\[
\land \sigma \models \partial \text{self}. \text{right}^{(1)} \land \sigma \models (\text{self}. \text{right}^{(1)}) \in \text{NodeKindSet}
\]

and prove automatically by monotonicity of the approximation functions and their point-wise inclusion:

\[
\text{NodeTypeSet} \subseteq \text{NodeKindSet}
\]

This kind of theorems remains valid if we add further classes in a class system.

Now we relate class invariants of subtypes to class invariants of supertypes. Here, we use cast functions described in the previous section; we write \(\text{self}^{(1)}[\text{Node}]\) for the object \(\text{self}\) converted to the type \(\text{Node}\) of its superclass. The trick is done by defining a new approximation for an inherited class Cnode on the basis of the approximation function of the superclass:

\[
\text{CnodeF} \equiv \lambda \sigma. \lambda X.
\]

\[
\{ \text{self} \mid \text{self}^{(1)}[\text{Node}] \in (\text{NodeKindF } \sigma (\lambda \text{obj. obj}^{(1)}[\text{Node}] \setminus X)) \land \cdots \}
\]

where the \(\ldots\) stand for the constraints specific to the subclass and \(\setminus\) denotes the pointwise application.

Similar to \[4\] or \[21\] we can handle mutual-recursive datatype definitions by encoding them into a type sum. However, we already have a suitable type sum together with the needed injections and projections, namely our universe type with the make and get methods for each class. The only requirement is, that a set of mutual recursive classes must be introduced “in parallel,” i.e., as one extension of an existing universe.

These type sets have the usual properties that one associates with object-oriented type systems. Let \(\mathcal{E}_N (\mathcal{R}_N)\) be the characteristic type set (characteristic kind set) of a class \(N\) and let \(\mathcal{E}_C\) and \(\mathcal{R}_C\) be the corresponding type sets of a direct subclass of \(N\), then our encoding process proves formally that the characteristic type set is a subset of the kind set, i.e.:

\[
\sigma \models \text{self} \in \mathcal{E}_N \longrightarrow \sigma \models \text{self} \in \mathcal{R}_N.
\]

And also, that the kind set of the subclass is (after type cast) a subset of the type set (and thus also of the kind set) of the superclass:

\[
\sigma \models \text{self} \in \mathcal{R}_C \longrightarrow \sigma \models \text{self}^{(1)}[\text{Node}] \in \mathcal{E}_N.
\]
These proofs are based on co-inductions and involve a kind of bi-simulation of (potentially) infinite object structures. Further, these proofs depend on theorems that are already proven over the pre-defined types, e.g., Object. These proofs were done in the context of the initial universe $U^0$ and can be instantiated directly in the new universe without replaying the proof scripts; this is our main motivation for an extensible construction.

7 A Modular Proof-Methodology for OO Modeling

In the previous sections, we discussed a technique to build extensible object-oriented data models. Now we turn to the wider goal of an modular proof methodology for object-oriented systems and explore achievements and limits of our approach with respect to this goal. Two aspects of modular proofs over object-oriented models have to be distinguished:

1. the modular construction of theories over object-data models, and
2. a modular way to represent dynamic type information or storage assumptions underlying object-oriented programs.

With respect to the former, the question boils down to what degree extensions of class systems and theories built over them can be merged. With respect to the latter, we will show how co-inductive properties over the store help to achieve this goal.

Figure 1. Merging Universes
7.1 Non-overlapping Merges

The positive answer to the modularity question is that object-oriented data-model theories can be merged provided that the extensions to the underlying object-data models are non-overlapping. Two extensions are non-overlapping, if their set of classes including their direct parent classes are disjoint (see Figure 1a). In these cases, there exists a most general type instance of the merged object universe $U^3$ (the type unifier of both extended universes $U^2a$ and $U^2b$); thus, all theorems built over the extended universes are still valid over the merged universe (see Figure 1a). We argue that the non-overlapping case is the pragmatically more important one; for example, all libraries of the HOL-OCL system \cite{7} were linked to the examples in its substantial example suite this way. Without extensibility, the datatype package would have to require the recompilation of the libraries, which takes in the case of the HOL-OCL system about 20 minutes.

7.2 Handling Overlapping Merges

Unfortunately, there is a pragmatically important case in object-oriented modeling that will by considered as an overlap in our package. Consider the case illustrated in Figure 1b. Here, the parent class $A$ is in the class set of both extensions (including parent classes). The technical reason for the conflict is that the order of insertions of “son”-classes into a parent class is relevant since the type sum $\alpha + \beta$ is not a commutative type constructor.

In our encoding scheme of object-oriented data models, this scenario of extensions represents an overlap that the user is forced to resolve. One pragmatic possibility is to arrange an order of the extensions by changing the import hierarchy of theories producing overlapping extensions. This worst-case results in re-running the proof scripts of either $B$ or $C$—usually a matter of a minute. Another option is to introduce an empty class $B'$ and inherit $B$ from there. A further option consists in adding a mechanism into our package allowing to specify for a child-class the position in the insertion-order.

7.3 Modularity in an Open-world: Dynamic Typing

Our notion of extensible class systems generalizes the distinction “open-world assumption” vs. “closed-world assumption” widely used in object-oriented modeling. Our universe construction is strictly “open-world” by default; the case of a “closed-world” results from instantiating all $\alpha, \beta$-“holes” in the universe by the unit type. Since such an instantiation can also be partial, there is a spectrum between both extremes. Furthermore, one can distinguish $\alpha$-finalizations, i.e., instantiation of an $\alpha$-variable in the universe by the unit type, and $\beta$-finalizations. The former close a class hierarchy with respect to subtyping, the latter prevent that a parent class may have further direct children (which makes the automatic derivation of an exhaustion lemma for this parent class possible).

Since methods can be overloaded, method invocations like in object-oriented languages require an overloading resolution mechanism such as late binding as
Late binding uses the dynamic type \( \text{isType}^{(1)}_X \) of \( \text{self} \). The late-binding method invocation is notorious for its difficulties for modular proof. Consider the case of an operation:

```verbatim
method Node::m():Bool
pre: P
post: Q
```

Furthermore assume that the implementation of \( m \) invokes itself recursively, e.g., by \( \text{self}.\text{left}.m() \). Based on an open-world assumption, the postcondition \( Q \) cannot be established in general since it is unknown which concrete implementation is called at the invocation.

Based on our universe construction, there are two ways to cope with this underspecification. First, finalizations of all child classes of \( \text{Node} \) results in a partial closed-world assumption allowing to treat the method invocation as case switch over dynamic types and direct calls of method implementations. Second, similarly to the co-inductive invariant example in Section 6 which ensures that a specific de-referentiation is in fact defined, it is possible to specify that a specific de-referentiation \( \text{self}.\text{left}^{(1)} \) has a given dynamic type. An analogous invariant \( \text{Inv}_{\text{left}}(\text{self}) \) can be defined co-inductively. From this invariant, we can directly infer facts like \( \text{isType}^{(1)}_{\text{Node}}(\text{self}.\text{left}^{(1)}) \), and \( \text{isType}^{(1)}_{\text{Node}}(\text{self}.\text{left}^{(1)}.\text{left}^{(1)}) \), i.e., in an object graph satisfying this invariant, the left “spine” must consist of nodes of dynamic type \( \text{Node} \). Strengthening the precondition \( P \) by \( \text{Inv}_{\text{left}}(\text{self}) \) consequently allows to establish postcondition \( Q \)—in a modular manner, since only the method implementation above has to be considered in the proof. Invoking the method on an object graph that does not satisfy this invariant can therefore be considered as a breach of the contract.

### 7.4 Modularity in an Open-world: Storage Assumptions

Similarly to co-inductive invariants, it is possible via co-recursive functions to map an object to the set of objects that can be reached along a particular path set. The definition must be co-recursive, since object structures may represent a graph. However, the presentation of this function may be based on a primitive-recursive approximation function depending on a factor \( k :: \text{nat} \) that computes this object set only up to the length \( k \) of the paths in the path set.

- \( \text{ObjSet}_{\text{left}} \sigma \text{self} \sigma = \{ \} \)
- \( \text{ObjSet}_{\text{left}} k \sigma \text{self} \sigma = \text{if } \sigma \vdash \partial \text{self} \text{ then } \{ \} \)
  - \( \text{else } \{ \text{self} \} \cup \text{ObjSet}_{\text{left}} (k-1) (\text{self}.\text{left}^{(1)} \sigma) \sigma \)

The function \( \text{ObjSet}_{\text{left}} \text{self} \sigma \) can then be defined as limit

\[
\bigcup_{n \in \text{Nat}} \text{ObjSet}_{\text{left}} n \text{self} \sigma
\]

On the other hand, we can add an state invariant on our concept of state per type definition \( \alpha \text{St} = \{ \sigma :: \text{oid} \rightarrow \mathcal{V}^\alpha. \text{Inv} \sigma \} \). Here, we require for \( \text{inv} \) that
each oid points to an object that contains itself:

\[ \forall \text{oid} \in \text{dom } \sigma. \text{OidOf(the}(\sigma \text{ oid})) = \text{oid} \]

As a consequence, there exists a “one-to-one”-correspondence between objects and their oid in a state. Thus, sets of objects can be viewed as sets of references, too, which paves the way to interpret these reference sets in different states and specify that an object did not change during a system transition or that there are no references from one object-structure into some critical part of another object structure.

8 Conclusion

We presented an extensible universe construction supporting object-oriented features such as subtyping and (single) inheritance. The underlying mapping from object-language types to types in the HOL representation is injective, which implies type-safety. We introduce co-inductive properties on object systems via characteristic sets defined by greatest fixed-points; these sets also give a semantics for class invariants. In our package, constructors and update-operations were handled too, but not discussed due to space limitations.

The universe-construction is supported by a package (developed as part of the HOL-OCL project [7]). Generated theories on object systems can be applied for object-oriented specification languages as OCL as well as programming language embeddings using the type-safe shallow technique.

In the context of HOL-OCL, we gained some experimental data that shows the feasibility of the technique: Table 1 describes the size of each of the above mentioned models together with the number of generated theorems and the time needed for importing them into HOL-OCL. The number of generated theorems depends linearly on the number of classes, attributes, associations, operations and OCL constraints. For generalizations, a quadratic number (with respect to the number of classes in the model) of casting definitions have to be generated and also a quadratic number of theorems have to be proven. The time for encoding the models depends on the number of theorems generated and also on the complexity on their complexity.

<table>
<thead>
<tr>
<th></th>
<th>Invoice</th>
<th>eBank</th>
<th>Company</th>
<th>Royals and</th>
<th>Loyals</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of classes</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>size of OCL specification (lines)</td>
<td>149</td>
<td>114</td>
<td>210</td>
<td>520</td>
<td></td>
</tr>
<tr>
<td>generated theorems</td>
<td>647</td>
<td>1444</td>
<td>1312</td>
<td>2516</td>
<td></td>
</tr>
<tr>
<td>time needed for import (in seconds)</td>
<td>12</td>
<td>42</td>
<td>49</td>
<td>136</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Importing Different UML/OCL Specifications.
8.1 Related Work

Datatype packages have been considered mostly in the context of HOL or functional programming languages. Systems like [14,4] build over a S-expression like term universe (co)-inductive sets which are abstracted to (freely generated) datatypes. Paulson’s inductive package [21] also uses subsets of the ZF set universe \(i\).

Work on object-oriented semantics based on deep embeddings has been discussed earlier. For shallow embeddings, to the best of our knowledge, there is only [22]. In this approach, however, emphasis is put on a universal type for the method table of a class. This results in local “universes” for input and output types of methods and the need for reasoning on class isomorphisms. Subtyping on objects must be expressed implicitly via refinement. With respect to extensibility of data-structures, the idea of using parametric polymorphism is partly folklore in HOL research communities; for example, extensible records and their application for some form of subtyping has been described in HOOL [17]. Since only \(\alpha\)-extensions are used, this results in a restricted form of class types with no cast mechanism to \(\alpha\) Object.

The underlying encoding used by the loop tool [11] and Jive [15] shares same basic ideas with respect to the object model. However, the overall construction based on a closed-world assumption and thus, not extensible. The support for class invariants is either fully by hand or axiomatic.

8.2 Future Work

We see the following lines of future research:

- **Towards a Generic Package.** The supported type language as well as the syntax for the co-induction schemes is fixed in our package so far. More generic support for the semantic infrastructure of other target languages is required to make our package more widely applicable.

- **Support of built-in Co-recursion.** Co-recursion can be used to define e. g., deep object equalities.

- **Deriving VCG.** Similar to the IMP-theory in the Isabelle library, Hoare-calculi for object-oriented programs can be derived (as presented in [8]). On this basis, verification condition generators can be proven sound and to a certain extent, complete. This leads to effective program verification techniques based entirely on derived rules.

References


A declarative language for the Coq proof assistant.*

Pierre Corbineau

pierre.corbineau@cs.ru.nl
Institute for Computing and Information Science
Radboud University Nijmegen, Postbus 9010
6500GL Nijmegen, The Netherlands

Abstract. This paper presents a new proof language for the Coq proof assistant. This language uses the declarative style. It aims at providing a simple, natural and robust alternative to the existing \texttt{Ltac} tactic language. We give the syntax of our language, an informal description of its commands and its operational semantics. We explain how this language can be used to implement formal proof sketches. Finally, we present some extra features we wish to implement in the future.

1 Introduction

1.1 Motivations

An interactive proof assistant can be described as a state machine that is guided by the user from the ‘statement \( \phi \) to be proved’ state to the ‘QED’ state. The system ensures that the state transitions (also known as proof steps in this context) are sound. The user’s guidance is required because automated theorem proving in any reasonable logics is undecidable in theory and difficult in practice. Therefore, an input language has to be used to guide the proof assistant.

A seminal work in this domain was the ML language developed for the LCF theorem prover [GMW79]. The ML language was a fully-blown programming language with specific functions called tactics to modify the proof state. The tool itself consisted of an interpreter for the ML language. Thus a formal proof was merely a computer program. With this in mind, think of a person reading somebody else’s formal proof, or even one of his/her own proofs but after a couple of months. Similarly to what happens with source code, this person will have a lot of trouble understanding what is going on with the proof unless he/she has a very good memory or the proof is thoroughly documented. Of course, running the proof through the prover and looking at the output might help a little.

This example illustrates a big inconvenient which still affects many popular proof languages used nowadays: they lack readability. Most proofs written are actually write-only or rather write- and execute-only, since what the user is interested when re-running the proof in is not really the input, but rather the

* This work was partially funded by NWO Bricks/Focus Project 642.000.501
output of the proof assistant, i.e. the sequence of proof states from stating the theorem to ‘QED’.

The idea behind declarative style proofs is to actually base the proof language on this sequence of proof states. This is indeed this very feature that makes the distinction between procedural proof languages (like ML tactics in LCF) and declarative proof languages. On the one hand, procedural languages emphasize proof methods (application of theorems, rewriting, proof by induction...), at the expense of a loss of precision on intermediate proof states. On the other hand, declarative languages emphasize proof states but are less precise about what allows to jump from one state to the next one.

1.2 Related work

The first proof assistant to implement a declarative style proof language was the Mizar system, whose modern versions date back to the early 1990’s. The Mizar system is a batch style proof assistant: it compiles whole files and writes error messages in the body of the input text, so it is not exactly interactive, but its proof language has been an inspiration for all later designs [WW02].

Another important source in this subject is Lamport’s How to write a proof [Lam95] which takes the angle of the mathematician and provides a very simple system for proof notations, aimed at making proof verification as simple as possible.

The first interactive theorem prover to actually provide a declarative proof language has been Isabelle [Law94], with the Isar (Intelligible Semi-Automated Reasoning) language [Wen99], designed by Markus Wenzel. This language has been widely adopted by Isabelle users since.

John Harrison has also developed a declarative proof language for the HOL88 theorem prover [Har96]. Freek Wiedijk also developed a light declarative solution [Wie01] for John Harrison’s own prover HOL Light [Har06].

For the Coq proof assistant [Coq07], Mariusz Giero and Freek Wiedijk have built a set of tactics called the MMode [GW04] to provide an experimental mathematical mode which give a declarative flavor to Coq.

Recently, Claudio Sacerdoti-Coen added a declarative language to the Matita proof assistant [Coe06].

1.3 A new language for the Coq proof assistant

The Coq proof assistant is a Type Theory-based interactive proof assistant developed at INRIA. It has a strong user base both in the field of software and hardware certification and in the field of formalized mathematics. It also has the reputation of being a tool whose procedural proof language \( \text{L} \text{tac} \) has a very steep learning curve both at the beginner and advanced level.

Coq has been evolving quite extensively during the last decade, and the evolution has made it necessary to regularly update existing proofs to maintain compatibility with most recent versions of the tool.
Coq also has a documentation generation tool to do hyper-linked rendering of files containing proofs, but most proofs are written in a style that makes them hard to understand (even with colored keywords) unless you can actually run them, as was said earlier for other procedural proof languages.

Building on previous experience from Mariusz Giero, we have built a stable mainstream declarative proof language for Coq. This language was built to have the following characteristics:

**readable** The designed language should use clear English words to make proof reading a feasible exercise.

**natural** We want the language to use a structure similar to the ones used in textbook mathematics (e.g. for case analysis), not a bare sequence of meaningless commands.

**maintainable** The new language should make it easy to upgrade the prover itself: errors in the proof should not propagate.

**stand-alone** The proof script should contain enough explicit information to be able to retrace the proof path without running Coq.

The Mizar language has been an important source of inspiration in this work but additional considerations had to be taken into account because the Calculus of Inductive Constructions (CIC) is much richer than Mizar’s (essentially) first-order Set Theory.

One of the main issue is that of proof generivity: Coq proofs use a lot of Inductive objects for lots of different applications (logical connectives, records, natural numbers, algebraic data-types, inductive relations...). Rather than enforcing the use of the most common inductive definitions, we want to be as generic as possible in the support we give for reasoning with these objects.

Finally, we want to give an extended support for proofs by induction by allowing multi-level induction proofs, using a very natural syntax to specify the different cases in the proof. The implementation was part of the official 8.1 released version of Coq.

### 1.4 Outline

We first describe some core features of our language, such as forward and backward steps, justifications, and partial conclusions. Then we give a formal syntax and a quick reference of the commands of our language, as well as an operational semantics. We go on by explaining how our language is indeed an implementation of the *formal proof sketches* [Wie03] concept, and we define the notion of well-formed proof. We finally give some perspective for future work.

### 2 Informal description

#### 2.1 An introductory example

To give a sample of the declarative language, we provide here the proof of a simple lemma about Peano numbers: the **double** function is defined by \( \text{double} \ x = x + x \).
and the \( \text{div}_2 \) functions by:

\[
\begin{align*}
\text{div}_2 0 &= 0 \\
\text{div}_2 1 &= 0 \\
\text{div}_2 (S (S x)) &= S (\text{div}_2 x)
\end{align*}
\]

The natural numbers are defined by means of an inductive type \text{nat} with two constructors 0 and \( S \) (successor function. The lemma states that \( \text{div}_2 \) is the left inverse of \text{double}. We first give a proof of the lemma using the usual tactic language:

\[
\text{Lemma double_div2: forall } n, \text{div}_2 (\text{double } n) = n.
\]
\text{intro } n. \\
\text{induction } n. \\
\text{reflexivity.} \\
\text{unfold double in } *|-*.
\text{simpl.} \\
\text{rewrite } <- \text{plus_n_Sm.} \\
\text{rewrite IHn.} \\
\text{reflexivity.} \\
\text{Qed.}
\]

Now, we give the same proof using the new declarative language:

\[
\text{Lemma double_div2: forall } n, \text{div}_2 (\text{double } n) = n.
\]
\text{proof.} \\
\text{let } n:\text{nat.} \\
\text{per induction on } n. \\
\text{suppose it is } 0. \\
\text{reconsider thesis as } (0=0). \\
\text{thus thesis.} \\
\text{suppose it is } (S \, m) \text{ and Hrec:thesis for } m. \\
\text{have \( (\text{div}_2 (\text{double } (S \, m)) = \text{div}_2 (S (\text{div}_2 (\text{double } m))) \)).} \\
\text{thus } = (S (\text{div}_2 (\text{double } m))). \\
\text{end induction.} \\
\text{end proof.} \\
\text{Qed.}
\]

The proof consists of a simple induction on the natural number \( n \). The first case is done by conversion (computation) to 0 = 0 and the second case by computation and by rewriting the induction hypothesis. Of course, you could have guessed that by simply reading the declarative proof.
2.2 Forward and backward proofs

The notions of declarative and procedural proofs are often confused with the notions of forward and backward proof. We believe those two notions are mostly orthogonal: the distinction between declarative and procedural is that declarative proofs mention explicitly the intermediate proof steps, while procedural proofs explain what method is used to go to the next state without mentioning it, whereas the distinction between forward and backward proof is merely a notion of whether the proof is build bottom-up by putting together smaller proofs, or top-down by cutting a big proof obligation into smaller ones.

The reason for the confusion between declarative style and backwards proofs is that most declarative languages rely on the core command

\texttt{have h : \phi \textit{justification}}

which introduces a new hypothesis \( h \) that asserts \( \phi \), after having proved it using \textit{justification}. This kind of command is the essence of forward proofs: it builds a new object — a proof of \( \phi \) — from objects that already exists and are somehow mentioned in the justification.

In order to show that you can also use backwards steps in a declarative script, our language contains a command

\texttt{suffices H_1 and \ldots and H_n \textit{to show} G \textit{justification}}

which acts as the dual of the \texttt{have} construction: it allows to replace the current statement to prove by sufficient conditions with stronger statements (as explained by the justification). For example, you can use this command to generalize your thesis before stating a proof by induction.

2.3 Justifications

When using a command to deduce a new statement from existing ones, or to prove that some statements suffice to prove a part of the conclusion, you need the proof assistant to fill in the gap. The justification is a hint to tell the proof assistant which proof objects are to be used in the process, and how they shall be used.

In our language, justifications are of the form

\texttt{by \pi_1, \ldots, \pi_n \textit{using} t}

the \( \pi_k \) are proof objects which can be hypotheses names (variables) but also more complex terms like application of a general result \( H : (\forall x : \texttt{nat}, P x) \) to a specific object \( n : \texttt{nat} \) to get a proof \( (H n) : P n \). \( t \) is a optional tactic expression that will be used to prove the validity of the step.

The meaning of the \( \pi_k \) objects is that only they and their dependencies — if \( H \) is specified and \( H \) has type \( P x \) then \( x \) is also implicitly added — will be in the local context in which the new statement is to be proved. If they are omitted
then the new statement should be either a tautology or provable without any
local hypothesis by the means of the provided tactic. If the user types by *,
all local hypotheses can be used. The use of by * should be transient because
it makes the proof more brittle: the justification depends too much on what
happened before.

If specified, the tactic t will then be applied to the modified context and if
necessary, the remaining subgoals will be treated with the assumption tactic,
which looks for a hypothesis convertible with the conclusion. If using t is omitted
then a default automation tactic is used. This tactic is a combination of auto,
congruence congruence and firstorder [Cor03].

When writing a sequence of deduction steps, it often happens that a state-
ment is only used in the next step. In that case the statement might be anony-
mous. By using the then keyword instead of have, this anonymous statement
will be added to the list of proof objects to be used in that particular justification.

If a justification fails, the proof assistant will issue a warning

\textit{Warning: insufficient justification.}

rather than an error. This allows the user to write proofs from the outside in by
filling the gaps, rather than linearly from start to end. In the CoqIDE interface,
this warning is emphasized by coloring the corresponding command with an
orange background. This way, a user reading a proof script will immediately
identify where work still needs to be done.

2.4 Partial conclusion, split thesis

The idea of partial conclusions is that within a deduction chain, some steps are
actually sub-formulae of the current conclusion, so they can be used to remove
that sub-formula from that conclusion. For example, A is a partial conclusion
for $A \land B$, it is also a partial conclusion for $A \lor B$. In the latter case, choosing
to prove $A$ implies a choice in the proof we want to make, by proving $A$ rather
than $B$. In our language, the user can type the command

\texttt{thus G justification}

to provide a partial conclusion $G$ whose validity is proved using justification. If
$G$ is not proved, the usual warning is issued, but if $G$ is not a sub-formula of the
conclusion then an error occurs: the user is trying to prove the wrong thing.

More precisely, the notion of partial conclusion is a consequence of the defini-
tion of logical connectives by inductive types. We will look for partial conclusions
in the possible sub-terms of proof terms based on inductive type constructors.

In the case of the conjunction $A \land B$, if we have a proof $\pi$ of $A$, using the
pairing constructor \texttt{conj} we can build a proof \texttt{conj $\pi ?_1$}, where $?_1$ is a place-
holder for the remaining item to be proved (i.e. $B$). This way, we can introduce
the notion of remaining conclusions (see Fig. 1). The plural here is because,
unfortunately, it might occur that a partial conclusions causes the thesis to split
(i.e. several place-holders are needed in the partial proof). It might even happen
that a part of the split thesis depends on another: keep in mind that the existential quantifier is represented by a dependent pair. Finally, when using \( P2 \) as a partial conclusion for \( \exists x : \text{nat}, P x \), even though a placeholder for \( \text{nat} \) should remain, this placeholder has to be filled by 2 because of typing constraints.

Using this mechanism, we can allow the user to build the proof piece by piece, by providing partial conclusions, and at each step replacing the part of the thesis by the remaining parts to be proved. Parts of the thesis are given numbers and their type can be referred to by using \( \text{thesis} \[ n \] \) for the type of placeholder \( ?_n \).

The keyword \( \text{thesis} \) alone refers to the type of the unique placeholder when there is only one. When there are more than one place-holders, the thesis is said to be split.

A split thesis will induce some constraints on what the user can do, e.g. the user cannot perform a proof by cases or introduce a hypothesis. Therefore in some cases is is preferable to avoid those situation by giving all the conclusion at once.

3 Syntax and semantics

3.1 Syntax

The figure 2 gives the complete formal syntax of the declarative language. the unbound non-terminals are \( \text{id} \) for identifiers, \( \text{num} \) for natural numbers, \( \text{term} \) and \( \text{type} \) for terms and types of the Calculus of Inductive Constructions, \( \text{pattern} \) refers to a pattern for matching against inductive objects.

3.2 Commands description

```plaintext
proof.
...
end proof.  This is the outermost block of any declarative proof. If several subgoals existed when the proof command occurred, only the first one is proved
```
Fig. 2. Syntax for the declarative language

<table>
<thead>
<tr>
<th>intermediate step</th>
<th>conclusive step</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>with previous step</td>
</tr>
<tr>
<td>have</td>
<td>opens sub-proof</td>
</tr>
<tr>
<td>thus</td>
<td>iterated equality</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
</tr>
<tr>
<td>[id:]^7type</td>
</tr>
<tr>
<td>thesis</td>
</tr>
<tr>
<td>thesis[num]</td>
</tr>
<tr>
<td>thesis for id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
</tr>
<tr>
<td>[id:]^7type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
</tr>
<tr>
<td>[by [term[,term]*]^7 [using tactic]^7</td>
</tr>
</tbody>
</table>

Fig. 3. Synthetic classification of forward steps
in the declarative proof. If the proof is not complete when encountering end
proof, then the proof is closed all the same, but with a warning, and Qed or
Defined to save the proof will fail.

have h: \( \phi \) justification.
then h: \( \phi \) justification. This command adds a new hypothesis h of type \( \phi \) in
the context. If the justification fails, a warning is issued but the hypothesis
is still added to the context. The then variant adds the previous fact to the
list of objects used in the justification.

thus h: \( \phi \) justification.

hence h: \( \phi \) justification. These commands behave respectively like have and
then but the proof of \( \phi \) is used as a partial conclusion. This can end the
proof or remove part of the proof obligations. These commands fail if \( \phi \) is
not a sub-formula of the thesis.

claim h: \( \phi \).

end claim. This block contains a proof of \( \phi \) which will be named h after end
claim. If the subproof is not complete when encountering end claim, then
the subproof is still closed, but with a warning, and Qed or Defined to save
the proof later will fail.

focus on \( \phi \).

end focus. This block is similar to the claim block, except that it leads to
a partial conclusion. In a way, focus is to claim what thus is to have. This
comes handy when the thesis is split and one of its parts is an implication or
a universal quantification: the focus block will allow to use local hypotheses.

(\( \text{thus} \) \( \sim = \) t justification).
(\( \text{thus} \) = \( \text{t justification} \)). These commands can only be used if the last step
was an equality \( l = r \). t should be a term of the same type as \( l \) and \( r \). If
\( \sim = \) is used then the justification will be used to prove \( v = t \) and the new
statement will be \( l = t \). Otherwise, the justification will be used to prove
\( t = u \) and the new statement will be \( t = r \). When present, the thus keyword
will trigger a conclusion step.

assume G: \( \Psi \) ...and we have x such that H: \( \Phi \).

let x be such that H: \( \Phi \).

Those commands are two
different flavors for the introduction of hypothesis. They expect the thesis
not to be split, and of the shape \( \Pi_i x_i : T_i G_i \). It expects the \( T_i \) to be
convertible with the provided hypotheses statements. This command is well-
formed only if the missing types can be inferred.

given x such that H: \( \Phi \).

consider x such that H: \( \Phi \) from G. given is similar to let, except that
this command works up to elimination of tuples and dependent tuples such as
conjunctions and existential quantifiers. Here the thesis could be \( \exists x \Phi' \to \Psi \)
with \( \Phi' \) convertible to \( \Phi \). The consider command takes an explicit object \( C \)
to destruct instead of using an introduction rule.

define f (x:T) as H. This command allows to defines objects locally. if pa-
rameters are given, a function (\( \lambda \)-abstraction) is defined.
reconsider thesis as $T$.
reconsider $H$ as $T$. These commands allow to replace the statement of a conclusion or a hypothesis with a convertible one, and fails if the provided statement is not convertible.

take $t$. This command allows to do a partial conclusion using an explicit proof object. This is especially useful when proving an existential statement: it allows to specify the existential witness.

per cases on $t$.
— of $F$ justification.
suppose $x : H$.
...
suppose $x' : H'$.
...
end cases.

This introduces a proof per cases on a disjunctive proof object $t$ or a proof of the statement $F$ derived from justification. The per cases command must immediately be followed by a suppose command which will introduce the first case. Further suppose commands or end cases can be typed even if the previous case is not complete. In that case a warning is issued. This block of commands cannot be used when the thesis is split. If $t$ occurs in the thesis, you should use suppose it is instead of suppose.

per induction on $t$.
— cases —
suppose it is $patt$ and $x : H$.
...
suppose it is $patt'$ and $x' : H'$.
...
end cases.

This introduces a proof per dependent cases or by induction. When doing the proof, $t$ is substituted with $patt$ in the thesis. $patt$ must be a pattern for a value of the same type as $t$. It may contain arbitrary sub-patterns and as statements to bind names to sub-patterns. Those name aliases are necessary to apply the induction hypothesis at multiple levels. If you are doing a proof by induction, you may use the thesis for construction in the suppose it is command to refer to induction hypothesis. You may also write induction hypotheses explicitly.

escape.
...
return. This block allows to escape the declarative mode back to the tactic mode. If the thesis is split, then several subgoals are provided, or the command fails if some parts depend on others.

3.3 Operational semantics

The purpose of this section is to give precise details about what happens to the proof state when you type a proof command. The proof state consists of a stack $S$ that contains open proofs and markers to count open sub-proofs, and
each sub-proof is a judgement $\Gamma \vdash \Delta$, where $\Gamma$ and $\Delta$ are lists of types (or propositions) indexed by names for $\Gamma$ and integers for $\Delta$. The intuition is that we want a proof of the conjunction of the types in $\Delta$ from the hypotheses in $\Gamma$. The rules are to be seen as stack transformations from top to bottom, with some side conditions.

When $\Delta$ is empty, the sub-proof is complete and the user has to close the block. If at some point $\Delta$ is supposed to be empty and is not, then the command will issue a warning.

The $j \triangleright \Gamma \vdash G$ expression means that the justification $j$ is sufficient to solve the problem $\Gamma \vdash G$. If it isn’t, the command issues a warning. The $\equiv$ relation is the conversion relation of the Calculus of Inductive Constructions.

The $L \sqsubseteq R$ relation tells us that the $L$ signature can be obtained by decomposing tuples in the $R$ signature. The $-$ operator computes the remaining parts of the thesis when a sub-formula is already proved. The details of the $-$ are too technical to be written here. We use the traditional $\lambda$ notation for abstractions and $\Pi$ for dependent products (either implication or universal quantification, depending on the context).

The distinction between cases$_d$ and cases$_{nd}$ is used to prevent the mixing of suppose with suppose it is. The next pages contain the transition rules. The coverage condition for case analysis has been omitted for simplicity.

\[
\begin{align*}
\frac{\{\Gamma \vdash G\}\text{[procedural]}}{(\Gamma \vdash ?_1 : G); []} \text{ proof.} & \quad \frac{(\Gamma \vdash \emptyset); []}{\text{(end of declarative proof)} \text{ end proof}.} \\
\frac{\{\Gamma \vdash \Delta\}; S \quad j \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{(\Gamma; x : T \vdash \Delta); S} & \quad \text{have} (x : T) \ j. \\
\frac{(\Gamma; l : T' \vdash \Delta); S \quad j, l \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{(\Gamma; l : T'; x : T' \vdash \Delta); S} & \quad \text{then} (x : T) \ j. \\
\frac{\{\Gamma \vdash \Delta\}; S \quad j \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{(\Gamma; x : T \vdash \Delta - (x : T)); S} & \quad \text{thus} (x : T) \ j. \\
\frac{(\Gamma; l : T' \vdash \Delta); S \quad j, l \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{(\Gamma; l : T'; x : T' \vdash \Delta - (x : T)); S} & \quad \text{hence} (x : T) \ j. \\
\frac{\{\Gamma \vdash \Delta\}; S \quad j \triangleright \Gamma \vdash r = u \quad \Delta \neq \emptyset}{(\Gamma; e : l = u \vdash \Delta); S} & \quad \text{=} u \ j. \\
\frac{(\Gamma \vdash \Delta); S \quad j \triangleright \Gamma \vdash u = l \quad \Delta \neq \emptyset}{(\Gamma; e : u = r \vdash \Delta); S} & \quad \text{=} u \ j. \\
\frac{(\Gamma \vdash \Delta); S \quad j \triangleright \Gamma \vdash r = u \quad \Delta \neq \emptyset}{(\Gamma; e : l = u \vdash \Delta - (l = u)); S} & \quad \text{thus} \text{=} u \ j. \\
\frac{(\Gamma \vdash \Delta); S \quad j \triangleright \Gamma \vdash u = l \quad \Delta \neq \emptyset}{(\Gamma; e : u = r \vdash \Delta - (u = r)); S} & \quad \text{thus} \text{=} u \ j.
\end{align*}
\]
\[(\Gamma \vdash \Delta); S \quad \Delta \neq \emptyset \quad \text{claim } (x : T).\]

\[(\Gamma \vdash \Delta)\vdash 1 : T; \text{claim}; (\Gamma; x : T \vdash \Delta); S \quad \text{end claim.}\]

\[(\Gamma \vdash \Delta); S \quad \Delta \neq \emptyset \quad (\Gamma \vdash ?1 : T); \text{focus}; (\Gamma; x : T \vdash \Delta \land (x : T)); S \quad \text{focus on } (x : T).\]

\[(\Gamma \vdash \Delta); S \quad \Delta \neq \emptyset \quad (\Gamma \vdash \Delta)\vdash 0; \text{focus}; (\Gamma \vdash \Delta); S \quad \text{end focus.}\]

\[(\Gamma \vdash \Delta); S \quad \Delta \neq \emptyset \quad (\Gamma \vdash \Delta \land (t : T)); S \quad \text{take } t.\]

\[(\Gamma \vdash \Delta); S \quad \Gamma; x_1 : T_1, \ldots, x_n : T_n \vdash t : T \quad \Delta \neq \emptyset \quad (\Gamma; \text{define } f (x_1 : T_1) \ldots (x_n : T_n) \text{ as } t).\]

\[(\Gamma \vdash ?p : \Pi x_1 : T_1 \ldots \Pi x_n : T_n. G); S \quad (T_1 \ldots T_n) \equiv (T_1' \ldots T_n') \quad (\Gamma \vdash \Delta); S \quad \text{assume/let } (x_1 : T_1) \ldots (x_n : T_n).\]

\[(\Gamma \vdash ?p : \Pi x_1 : T_1' \ldots \Pi x_n : T_n'. G); S \quad (T_1' \ldots T_m) \subseteq (T_1' \ldots T_n') \quad (\Gamma \vdash ?p : \Pi x_1 : T_1' \ldots \Pi x_n : T_n'. G); S \quad \text{given } (x_1 : T_1) \ldots (x_m : T_m).\]

\[(\Gamma \vdash \Delta); S \quad \Gamma; t : T \quad (T_1 \ldots T_n) \subseteq (T) \quad \Delta \neq \emptyset \quad (\Gamma \vdash ?p : \Pi T); S \quad \text{consider } (x_1 : T_1) \ldots (x_n : T_n) \text{ from } t.\]

\[(\Gamma \vdash \Delta); S \quad T \equiv T' \quad \Delta \neq \emptyset \quad (\Gamma \vdash \Delta); S \quad \text{reconsider } x \text{ as } T.\]

\[(\Gamma \vdash \Delta); S \quad \text{reconsider thesis}[p] \text{ as } T.\]

\[(\Gamma \vdash \Delta); S \quad \text{per cases of } T \quad j \vdash \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash T \quad \Delta \neq \emptyset\]

\[(\Gamma \vdash \Delta \land T) \cup \{T_1; \ldots; T_n\}; S \quad \text{suffices } (x_1 : T_1) \ldots (x_n : T_n) \text{ to show } T \quad j.\]

\[\text{cases}_\text{nd}(t : T); (\Gamma; x : T \vdash ?p : G); S \quad \text{per cases of } T \quad j.\]

\[\text{cases}(t : T); (\Gamma \vdash \Delta); S \quad \text{per cases of } t.\]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{suppose } (x_1 : T_1) \ldots (x_n : T_n).\]

\[\text{cases}(t : T); (\Gamma \vdash \Delta); S \quad \text{suppose } (x_1 : T_1) \ldots (x_n : T_n).\]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{suppose it is } p \text{ and } (x_1 : T_1) \ldots (x_n : T_n).\]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{cases}(t : T); (\Gamma \vdash ?p : G); S \]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{cases}(t : T); (\Gamma \vdash ?p : G); S \]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{cases}(t : T); (\Gamma \vdash ?p : G); S \]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{cases}(t : T); (\Gamma \vdash ?p : G); S \]

\[\text{cases}(t : T); (\Gamma \vdash ?p : G); S \quad \text{cases}(t : T); (\Gamma \vdash ?p : G); S \]
suppose it is $p$ and $(x_1:T_1)\ldots(x_n:T_n)$.

\[ (\Gamma \vdash \emptyset); \text{induction}(t:T); (\Gamma \vdash ?_p : G); S \]

end induction.

4 Proof edition

4.1 Well-formedness

If we drop the justification and completeness conditions in our formal semantics, we get a notion of well-formed proofs. Those proofs when run in Coq, are accepted with warnings but cannot be saved for they contain dummy proofs at some places of the proof tree, to fill in the gaps that the justification could not fill.

This does not prevent the user from going further with the proof since the user is still able to use the result from the previous step. The smallest well formed proof is:

\begin{verbatim}
proof.
end proof.
\end{verbatim}

Introduction steps such as assume have additional well-formedness requirements: the introduced hypotheses must match the available ones. The given construction allows a looser correspondence. The reconsider statements have to give a convertible type.

For proofs by induction, well-formedness requires the patterns to be of the correct type, and induction hypotheses to be build from the correct sub-objects in the pattern.
4.2 Formal proof sketches

We claim that well-formed but incomplete proofs in our language play the role of formal proof sketches: they ensure that hypotheses correspond to the wanted statement and that object referred to exists and have a correct type. When avoiding the by * construction, justifications are preserved when adding extra commands inside the proof. In this sense our language supports incremental proof development.

The only thing that the user might have trouble doing when turning a textbook proof into a proof sketch in our language is ensuring that first-order objects are introduced before a statement refers to them. Once this is done, the user will be able to add new lines within blocks (mostly forward steps).

5 Conclusion & Further work

5.1 Further work

*Arbitrary relation composition* The first extension that is needed for our language is the support for iterated relations other than equality. This is possible as soon as a generalized transitivity lemma of the form \( \forall xyz, x R_1 y \rightarrow y R_2 z \rightarrow x R_3 z \) is available.

*Better automation* There is a need for a more precise and powerful automation for the default justification method, to be able to give better predictions of when a deduction step will be accepted. A specific need would be an extension of equality reasoning to arbitrary equivalence relations (setoids, PERs . . .).

*Multiple induction* The support for induction is already quite powerful (support for deep patterns with multiple as bindings), but more can be done if we start considering multiple induction. It might be feasible to detect the induction scheme used (double induction, lexicographic induction ...) to build the corresponding proof on-the-fly.

*Translation of procedural proofs* The declarative language offers a stable format for the preservation of old proofs over time. Since many Coq proofs in procedural style already exist, it will be necessary to translate them to this new format. The translation can be done in two ways: by generating a declarative proof either from the proof tree, or from the proof term. The latter will be more fine grain but might miss some aspects of the original procedural proof. The former looks more difficult to implement.

5.2 Conclusion

The new declarative language is now widely distributed and we hope that this paper will help new users to discover our language. The implementation is quite
stable and the automation, although not very predictable, offers a reasonable compromise between speed and power.

We really hope that this language will be a useful medium to make proof assistant more popular, especially in the mathematicians community and among undergraduate students. We believe that our language provides a helpful implementation of the formal proof sketch concept; this means it could be a language of choice for turning textbook proofs into formal proofs. It could also become a tool of choice for education.

In the context of collaborative proof repositories (using the Wiki paradigm), our language, together with other declarative languages, will fill the gap between the narrow proof assistant community and the general public: we aim at presenting big formal proofs to the public.

References


Isabelle Theories for Machine Words

Jeremy E. Dawson\(^1\,\,2\)

\(^1\) Logic and Computation Program, NICTA \(^*\)
\(^2\) Automated Reasoning Group,
Australian National University, Canberra, ACT 0200, Australia
http://users.rsise.anu.edu.au/~jeremy/

Abstract. We describe a collection of Isabelle theories which facilitate reasoning about machine words. For each possible word length, the words of that length form a type, and most of our work consists of generic theorems which can be applied to any such type. We develop the relationships between these words and integers (signed and unsigned), lists of booleans and functions from index to value, noting how these relationships are similar to those between an abstract type and its representing set. We discuss how we used Isabelle’s \texttt{bin} type, before and after it was changed from a datatype to an abstract type, and the techniques we used to retain, as nearly as possible, the convenience of primitive recursive definitions. We describe other useful techniques, such as encoding the word length in the type.

Keywords: machine words, twos-complement, mechanised reasoning

1 Introduction

In formally verifying machine hardware, we need to be able to deal with the properties of machine words. These differ from ordinary numbers in that, for example, addition and multiplication can overflow, with overflow bits being lost, and there are bit-wise operations which are simply defined in a natural way.

Wai Wong [8] developed HOL theories in which words are represented as lists of bits. The type is the set of all words of any length; words of a given length form a subset. Some theorems have the word length as an explicit condition. The theories include some bit-wise operations but not the arithmetic operations.

In [4] Fox describes HOL theories modelling the architecture of the ARM instruction set. There, the HOL datatype \texttt{w32 = W32 of num} is used, that is, the machine word type is isomorphic to the naturals, and the expression \texttt{W32 n} is to mean the word with unsigned value \(n\ mod\ 2^{32}\). In this approach, equality of machine words does not correspond to equality of their representations.

In [1] Akbarpour, Tahar & Dekdouk describe the formalisation in HOL of fixed point quantities, where a single type is used, and the quantities contain

\(^*\) National ICT Australia is funded by the Australian Government’s Dept of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia’s Ability and the ICT Centre of Excellence program.
fields showing how many bits appear before and after the point. Their focus is on the approximate representation of floating point quantities.

In [5] Harrison describes the problem of encoding vectors of any dimension $n$ of elements of type $A$ (e.g. reals, or bits) in the type system of HOL, the problem being that a type cannot be parameterised over the value $n$. His solution is to use the function space type $N \rightarrow A$, where $N$ is a type which has exactly $n$ values. He discusses the problem that an arbitrary type $N$ may in fact have infinitely many values, when infinite dimensional vectors are not wanted.

In the bitvector library [2] for PVS, which has a more powerful type system, a bit-vector is defined as a function from $\{0, \ldots, N-1\}$ to the booleans. Interpretations of a bit-vector as unsigned or signed integers, with relevant theorems, are provided in that library.

In this paper we describe theories for Isabelle/HOL [6] developed for reasoning about machine words for NICTA’s L4.verified project [7], which aims to provide a mathematical, machine-checked proof of the conformance of the L4 microkernel to a high level, formal description of its expected behaviour. As in [5], each type of words in our formalization is of a particular length. In this work we relate our word types both to the integers modulo $2^n$ and to lists of booleans; thus we have access to large bodies of results about both arithmetic and logical (bit-wise) operations. We have defined all the operations referred to in [4], and describe several other techniques and classes of theorems.

Our theories have been modified recently due to our collaboration with the company Galois Connections, who have developed similar, though less extensive, theories. The Galois theories, though mostly intended to be used for $n$-bit machine words, are based on an abstract type of integers modulo $m$ (where, for machine words, $m = 2^n$). Thus, when we combined the theories recently, we used the more general Galois definition of the abstract type $\alpha$ word; our theorems apply when $\alpha$ belongs to an axiomatic type class for which $m = 2^n$.

In this paper we focus on the techniques used to define the class. We defined numerous operations on words which are not discussed here, such as concatenating, splitting, rotating and shifting words. Some of these are mentioned in the Appendix. The Isabelle code files are available at [3].

2 Description of the word-n theories

2.1 The bin and obin types

The bin type explicitly represents bit strings, and is important because

- it is used for encoding literal numbers, and an integer entered in an Isabelle expression is converted to a bin, thus read "3" gives


  (where $x :: T$ means that $x$ is of type $T$);

- there is much built-in numeric simplification for numbers expressed as bins, for example for negation, addition and multiplication, using rules which reflect the usual definitions of these operations for the usual twos-complement representation of integers.
Isabelle changed during development of our theories. Formerly the bin type was a datatype, with constructors

- Pls (a sequence of 0, extending infinitely leftwards)
- Min (a sequence of 1, extending infinitely leftwards) (for the integer $-1$)
- BIT (where $(w::bin) \text{ BIT } (b::bool)$ is $w$ with $b$ appended on the right)

Subsequently, in Isabelle 2005, Isabelle’s bin type changed. The new bin type in Isabelle 2005 is an abstract type, isomorphic to the set of all integers, with abstraction and representation functions Abs_Bin and Rep_Bin.

We found that each of these ways of formulating the bin type has certain advantages. We proceed to discuss these, and how we overcame the disadvantages of the new way of defining bins. So we first describe how we used the datatype-based definition.

Since at one stage in the course of adapting to this change we were using both the old and new definition of bins and associated theorems, we used new names for the old definition, with ‘o’ or ‘O’ prepended: thus we had the constructors oPls, oMin, OBiT, for the datatype obin. (We also kept the old function number_of, renaming it onum_of). So in describing our use of bins as formerly defined, we use these names.  

### 2.2 Definitions using the obin datatype

As these definitions have since been removed, this section is not relevant for using these theories currently. But we give this description to indicate the advantages and disadvantages of the obin type, i.e., the former, datatype-based definition of the bin type. In fact for some time we continued to use the obin type because it is defined as a datatype: only a datatype permits the primitive and general recursive definitions described below.

Using the obin datatype allows us to define functions in the most natural way in terms of their action on bits. For example, to define bit-wise complementation, we just used the following primitive recursive definitions:

```isabelle
primrec
obin_not_Pls : "obin_not oPls = oMin"
obin_not_Min : "obin_not oMin = oPls"
obin_not_OBIT : "obin_not (w OBIT x) = obin_not w OBIT Not x"
```

We mention that, apart from the obvious benefit of using a simple definition, it is easier to be sure that it accurately represents the action of hardware that we intend to describe: this is important in theories to be used in formal verification.

Defining bit-wise conjunction using primitive recursion on either of two arguments is conceptually similar, though the expression is not so simple.  

---

3 More recently, the bin type changed again, in development versions of Isabelle during 2006, to be identical to the integers rather than an isomorphic type. Now our references to the type bin indicate an integer expressed using Pls, Min and BIT.

4 In Isabelle a set of primitive recursive definitions must be based on the cases of exactly one curried argument. It can be easier to use Isabelle’s recdef package.
We also made considerable use of functions \texttt{obin\_last} and \texttt{obin\_rest}, which give the last bit and the remainder, respectively. Again, we defined these functions by primitive recursion using the fact that \texttt{obin} is a datatype (the rules correspond to the simplifications proved for \texttt{bin\_last} and \texttt{bin\_rest}, see §2.3).

In working with the \texttt{obin} type, we needed to define the concept of a normalized \texttt{obin}, where the combination \texttt{OPls OBIT False} does not appear, since it denotes the same sequence of bits, and so the same integer, as \texttt{OPls}. So we normalise an \texttt{obin} by changing \texttt{OPls OBIT False} to \texttt{OPls}, and likewise \texttt{OMin OBIT True} to \texttt{OMin}. Thus the set of normalised \texttt{obins} is isomorphic to the set of integers, via the usual two-complement representation (see theorems \texttt{td\_int\_obin} in §2.5, and \texttt{td\_ext\_int\_obin} in §2.6).

\begin{verbatim}
mk\_norm\_obin :: "obin => obin" is\_norm\_obin :: "obin => bool"
\end{verbatim}

While use of the \texttt{obin} type has the advantage over the \texttt{bin} type of being a datatype, the need to prove a large number of lemmas concerning normalisation of \texttt{obins} was a significant disadvantage.

### 2.3 Definitions involving the \texttt{bin} type

Our initial development developed words of length \(n\) from the set of \texttt{obins}. So, for example, we defined the bit-wise complement of a word using \texttt{obin\_not}, described above, and the addition of two words using addition of \texttt{obins}, based on functions to do numerical arithmetic from the Isabelle source files.

However we found the need to deal with words entered literally: \(6 \mapsto \text{'a word} is read as \text{number\_of} \langle\text{Pls BIT.B1 BIT.B1 BIT.B0}\rangle\). To simplify \(6 \&\& 5 \mapsto \text{'a word} (where \&\& is our notation for bit-wise conjunction), we found it convenient to use simplifications based on the \texttt{bin} type: that is, we wanted to use a function \texttt{bin\_and}, for bit-wise conjunction of \texttt{bins}, rather than \texttt{obin\_and}. Similarly, dealing with words of length 3, say, we wanted to simplify \(11 \mapsto \text{'a word} to 3\) using a function which truncates \texttt{bins}, not \texttt{obins}.

Since \texttt{bin} is not a datatype, we could not define functions on \texttt{bins} in the same way we did on \texttt{obins}. So, originally, we defined such functions on \texttt{bins} by reference to the corresponding functions on \texttt{obins}. To do this we used the functions \texttt{onum\_of} and \texttt{int\_to\_obin}, which relate the \texttt{int} (isomorphic to \texttt{bin}) and \texttt{obin} types.

\begin{verbatim}
bin\_and\_def : "bin\_and v w ==
onum\_of (obin\_and (int\_to\_obin v, int\_to\_obin w))"
\end{verbatim}

We had obtained a large number of simplification theorems involving \texttt{obins}. Using this approach, we then had to do some rather complex programming to transfer all these simplification theorems, \textit{en masse}, from \texttt{obins} to \texttt{bins}, so as to avoid proving them all again individually. In this way the parallel use of \texttt{obins} and \texttt{bins} produced significant extra complexity.

In short, we found that, although the fact of \texttt{obin} being a datatype permits simple recursive definitions, the machinery needed to take these definitions and
resulting theorems on "bins" and produce definitions and theorems for corresponding functions involving "bins" was unpleasantly cumbersome.

Therefore we examined alternative ways of defining functions in terms of the bit-representation of a "bin". First we considered what properties of the "bin" type resemble the properties of a datatype. The properties of a datatype are:

(a) Different constructors give distinct values
(b) Each constructor is injective (in each of its arguments)
(c) All values of the type are obtained using the constructors

Now we can consider the "bin" type with "pseudo-constructors" Pls, Min and Bit (where Bit w b is printed and may be entered as w BIT b).

In terms of these "pseudo-constructors" the properties (b) and (c) above hold: in fact property (c) holds using the "pseudo-constructor" Bit alone.

Thus we have these theorems; bin_exhaust enables us to express any "bin" appearing in a proof as w BIT b. Here "!!" is Isabelle notation for the universal quantification provided in the meta-logic.

BIT_eq = "u BIT b = v BIT c ==> u = v & b = c"
bin_exhaust = "(!!x b. bin = x BIT b ==> Q) ==> Q"

Then we can define functions bin_rl, and thence bin_last and bin_rest:

defs
bin_rl_def : "bin_rl w == SOME (r, l). w = r BIT l"
bin_rest_def : "bin_rest w == fst (bin_rl w)"
bin_last_def : "bin_last w == snd (bin_rl w)"

The meaning of the SOME function is that if there is a unique choice of r and l to satisfy w = r BIT l, then bin_rl (r BIT l) = (r, l). In fact property (b) gives this uniqueness, and so we can prove the expected simplification rules bin_last_simps and bin_rest_simps'. We then proved, as bin_last_mod and bin_rest_div, numerical characterisations of these functions.

bin_last_simps = "bin_last Pls = bit.B0 &
bin_last Min = bit.B1 & bin_last (w BIT b) = b"
bin_rest_simps' = "bin_rest Pls = Pls &
bin_rest Min = Min & bin_rest (w BIT b) = w"

bin_last_mod = "bin_last w == if w mod 2 = 0 then bit.B0 else bit.B1"
bin_rest_div = "bin_rest w == w div 2"

We also derived a theorem for proofs by induction involving "bins". While the premises of bin_induct contain some redundancy, this is unlikely to make a proof using bin_induct more difficult than it need be.

bin_induct = "[| P Pls; P Min;
!!bin bit. P bin ==> P (bin BIT bit) |] ==> P bin"
Both bin_exhaust and bin_induct were frequently used in proofs, and they usually made proofs for bins just as easy as the corresponding proofs for obins. Often the theorems and proofs were simpler for bins, e.g.

\begin{verbatim}
bin_add_not = "x + bin_not x = Min"
obin_add_not = "mk_norm_obin (obin_add x (obin_not x)) = oMin"
\end{verbatim}

However obtaining a near-equivalent, for bins, of primitive recursive definitions in obins, was a little more intricate. We have already described the definition of bin_last and bin_rest, and the derivation of simplification rules corresponding to the definitions of obin_last and obin_rest.

Typically a function \( f \) defined by primitive recursion would, if bin were a datatype with its three constructors, be defined by giving values \( v_P \) and \( v_N \) for \( f \text{ Pls} \) and \( f \text{ Min} \), and a function \( f_r \), where \( f \text{ (w BIT b)} \) is given by \( f_r \text{ w b (f w)} \). (The form of the recursion function returned by \textit{define_type} in the HOL theorem prover makes this explicit).

So we defined a function \textit{bin_rec} which, given \( v_P \), \( v_N \) and \( f_r \), returns a function \( f \) satisfying the three equalities shown, but the last only where \( w \text{ BIT b} \) does not equal Pls or Min.

\begin{verbatim}
bin_rec :: \"\text{\textit{a => \textquotesingle a => (int => bit => \textquotesingle a) => int => \textquotesingle a\textit{}}\"}
\end{verbatim}

\begin{verbatim}
f Pls = v_P
f Min = v_N
f (w BIT b) = f_r w b (f w)
\end{verbatim}

In the usual case, we can then prove that this last equation in fact holds for all \( w \) and \( b \), as we want for a convenient simplification rule. See examples in [3, \texttt{BinGeneral.thy}]. Here are \textit{bin_not} and \textit{bin_and} defined in this way:

\begin{verbatim}
defs
  bin_not_def : "bin_not == bin_rec Min Pls
  (\%w b s. s BIT bit_not b)"
 bin_and_def : "bin_and == bin_rec (\%x. Pls) (\%y. y)
  (\%w b s y. s (bin_rest y) BIT (bit_and b (bin_last y)))"
\end{verbatim}

After making these definitions, the simplification rules in the desired form (such as those shown below) need to be proved.

\begin{verbatim}
bin_not_simps = [... ,
  "bin_not (w BIT b) = bin_not w BIT bit_not b"
bin_and_simps = "bin_and (x BIT b) (y BIT c) =
  bin_and x y BIT bit_and b c"
\end{verbatim}

Proving these was virtually automatic for \textit{bin_not} (with one argument), and not difficult, but a little tedious, for \textit{bin_and} (with two arguments): see examples in [3, \texttt{BinGeneral.thy}]. However this turned out to be much easier than maintaining collections of corresponding theorems for the separate types \texttt{bin} and \texttt{obin}. 
2.4 The type of fixed-length words of given length

As a preliminary step, we define functions which create \( n \)-bit quantities. We called these “truncation” functions, although they also lengthen shorter quantities. Both functions will cut down a longer quantity to the desired length, by deleting high-order bits. For an argument shorter than desired, unsigned truncation extends it to the left with zeroes, whereas signed truncation extends it with its most significant bit. Thus \( \text{bintrunc} \ n \ w \) gives \( \text{Pls} \) followed by \( n \) bits, whereas \( \text{sbintrunc} \ (n-1) \ w \) (used for fixed-length words of length \( n \)) gives \( \text{Pls} \) or \( \text{Min} \) followed by \( n - 1 \) bits (so here the \( \text{Pls} \) or \( \text{Min} \), is treated as a sign bit, as one of the \( n \) bits). We defined \( \text{bintrunc} \) by primitive recursion on the first argument (the number of bits required) and auxiliary functions \( \text{bin_last} \) and \( \text{bin_rest} \), and \( \text{sbintrunc} \) similarly.

\[
\begin{align*}
\text{bintrunc}, \ \text{sbintrunc} & : \text{nat} \Rightarrow \text{bin} \Rightarrow \text{bin} \\
\text{primrec} & \\
\text{Z} : \text{"bintrunc} \ 0 \ \text{bin} = \text{Pls}" \\
\text{Suc} : \text{"bintrunc} \ (\text{Suc} \ n) \ \text{bin} = \\
& \text{bintrunc} \ n \ (\text{bin_rest} \ \text{bin}) \ \text{BIT} \ (\text{bin_last} \ \text{bin})"
\end{align*}
\]

Now we need to set up a type in which the length of words is implicit. The type system of Isabelle is similar to that of HOL in that dependent types are not allowed, so we cannot directly set up a type which consists of (for example) lists of length \( n \). Our solution was that the type of words of length \( n \) is \( \alpha \text{word} \) parametrised over the type \( \alpha \) where the word length can be deduced from the type \( \alpha \). As noted, Harrison did this by letting the word length be the number of values of the type \( \alpha \).

We use \( \text{len_of} \ \text{TYPE}(\alpha) \) for the word length. \( \text{TYPE}(\alpha) \) is a polymorphic value, of type \( \alpha \text{ itself} \), whose purpose is essentially to encapsulate a type as a term. In the output of \( \text{TYPE}(\alpha) \) the type \( \alpha \) is printed, which was useful. The function \( \text{len_of} \) is declared, with polymorphic type \( (\alpha, \text{printed as } 'a, \text{being a type variable}) \) in the library files as shown below. The library files provide the axiom \( \text{word_size} \) which gives the general formula for the length of a word, but the user must define the value of \( \text{len_of} \ \text{TYPE}(\alpha) \) for each specific choice of \( \alpha \).\(^5\)

\[
\begin{align*}
\text{len_of} & : \text{"'a :: len0 \ itself \Rightarrow \nat"} \\
\text{word_size} & : \text{"size (w :: 'a :: len0 \ word) \Rightarrow \text{len_of} \ \text{TYPE ('a)}"}
\end{align*}
\]

A type of fixed-length words is \( 'a :: \text{len0 \ word} \), where \( \text{len0} \) is a type class whose only relevance is that it admits a function \( \text{len_of} \), and the word length of any \( w :: 'a :: \text{len0 \ word} \) is given by the axiom \( \text{word_size} \). For each desired word length, the user declares a type (say \( a \)), in the class \( \text{len0} \), and defines the

\(^5\) Originally we used \( \text{len_of} \ (\text{arbitrary :: } \alpha) \) for the word length, but Isabelle doesn’t print the type of a constant such as \( \text{arbitrary} \), which was a difficulty in doing proofs involving different word lengths.
value `len_of TYPE (a)` to be the chosen word length. This provides a type of words of that given length.

(Isabelle notation may be confusing here: in `w :: 'a :: len0 word`, `w` is a term, `'a` is a type variable, `len0` is the type class to which `'a` belongs, and `word` is a type constructor. Thus the implicit bracketing is `w :: (('a :: len0) word)`.)

An Isabelle type definition defines a new type whose set of values is isomorphic to a given set. To define each word type we used the definition:

```isar
typedef 'a word = "uword_len (len_of TYPE ('a))"
  "uword_len len == range (bintrunc len)"
```

where `uword_len (len_of TYPE ('a))` is the set of integers, truncated to length `n` using the function `bintrunc` described earlier. 

The type class `len` is a subclass of `len0`, defined by the additional requirement that the word length `n` is non-zero.

`len_gt_0 = "0 < len_of TYPE('a::len)"
```

Results involving a signed interpretation of words are limited to this case (naturally, as the word needs to contain a sign bit). 

Thus the fixed-length word type is abstract, representing a sequence of bits, but such words can be interpreted as unsigned or signed integers. Although the abstract type is defined to be isomorphic to `range (bintrunc n)`, it can be viewed as isomorphic to several different sets. So the set of words of length `n` is isomorphic to each of the following, with the relevant “type definition theorems” (explained later) given in brackets:

- the set of integers in the range `0...2^n - 1 (td_uint)`
- the set of integers in the range `-2^{n-1}...2^{n-1} - 1 (td_sint)`
- the set of naturals up to `2^n - 1 (td_unat)`
- the set of lists of booleans of length `n (td_bl)`
- the set of functions `f` of type `nat -> bool` satisfying the requirement that for `i > n`, `f i = False (td_nth)`

That the type of a word implies its length had some curious consequences. For functions such as `ucast`, which casts a word from one length to another, or `word_rsplit`, which splits a word into a list of words of some given (usually shorter) length, the length of the resulting words is implicit in the result type of the function, not given as an argument. Therefore we get theorems such as "`ucast w = w" and "word_rsplit w = [w]", where the repeated use of the variable `w` implies that the result word(s) are of the same length as the argument.

---

6 This is the definition we used before combining our theories with those of Galois Connections, see §1.

7 Note that some other results are limited to `n > 0` because their proof uses theorems from the Isabelle library which apply only in a type class where `0` and `1` are distinct.
2.5 Pseudo type definition theorems

In Isabelle, defining a new type $\alpha$ from a set $S : \rho$ set causes the creation of an abstraction function $\text{Abs} : \rho \rightarrow \alpha$ and a representation function $\text{Rep} : \alpha \rightarrow \rho$, such that $\text{Abs}$ and $\text{Rep}$ are mutually inverse bijections between $S$ and the set of all values of type $\alpha$. Note that the domain of $\text{Abs}$ is the type $\rho$, but that nothing is said about the values it takes outside $S$. The predicate $\text{type_definition}$ expresses these properties, and a theorem, $\text{type_definition}_\alpha$, stating $\text{type_definition Rep Abs S}$, is created for the new type $\alpha$.

We can use the predicate $\text{type_definition}$ to express the isomorphisms between the set of $n$-bit words and the other sets mentioned above; we have proved the following “type definition theorems”:

- $\text{td_int_obin} = "\text{type_definition int_to_obin onum_of}
  (\text{range mk_norm_obin})"$
- $\text{td_uint} = "\text{type_definition uint word_of_int (uints (len_of TYPE('a)))}"$
- $\text{td_sint} = "\text{type_definition sint word_of_int (sints (len_of TYPE('a)))}"$
- $\text{td_unat} = "\text{type_definition unat of_nat (unats (len_of TYPE('a)))}"$
- $\text{td_bl} = "\text{type_definition to_bl of_bl}
  \{\text{bl::bool list. length bl = len_of TYPE('a)}\}"$
- $\text{td_nth} = "\text{type_definition word_nth of_nth}
  \{f::nat => \text{bool. ALL i::nat. f i --> i < len_of TYPE('a)}\}"$

These use the following functions between the various types (of_nat and onum_of have more general types, but are used with these types in these theorems):

- $\text{int_to_obin} :: "\text{int} \Rightarrow \text{obin}"
- \text{onum_of} :: "\text{obin} \Rightarrow \text{int}"
- \text{word_of_int} :: "\text{int} \Rightarrow 'a :: \text{len0 word}"
- \text{uint} :: "'a :: \text{len0 word} \Rightarrow \text{int}"
- \text{sint} :: "'a :: \text{len word} \Rightarrow \text{int}"
- \text{of_nat} :: "\text{nat} \Rightarrow 'a :: \text{len0 word}"
- \text{unat} :: "'a :: \text{len0 word} \Rightarrow \text{nat}"
- \text{of_bl} :: "\text{bool list} \Rightarrow 'a \text{ word}"
- \text{to_bl} :: "'a \text{ word} \Rightarrow \text{bool list}"
- \text{of_nth} :: "(\text{nat} \Rightarrow \text{bool}) \Rightarrow 'a \text{ word}"
- \text{word_nth} :: "'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{bool}"

The following define the representing sets referred to above, or were subsequently proved about them:

- "$\text{uints n == range (bintrunc n)}"$
- "$\text{sints n == range (sbintrunc (n - 1))}"
- "$\text{unats n == \{i. i < 2 \cdot n\}}"
- "$\text{uints n == \{i. 0 <= i \& i < 2 \cdot n\}}"
- "$\text{sints n == \{i. - (2 \cdot (n - 1)) <= i \& i < 2 \cdot (n - 1)\}}$"
2.6 Extended type definition theorems

As noted, however, these type definition theorems do not say anything about the action of \texttt{Abs} outside the set \texttt{S}. But in fact we have defined the abstraction functions to behave “sensibly” outside \texttt{S}, and it is useful to do so. For example, \texttt{word_of_int}, which turns an integer in the range \(0 \ldots 2^n - 1\) into a word, is defined so that it also behaves “sensibly” on other integers — it takes \(i\) and \(i'\) to the same word if \(i \equiv i' \pmod{2^n}\). This allows us to use the same abstraction function \texttt{word_of_int} in both theorems \texttt{td_uint} and \texttt{td_sint}.

\[\text{"word_of_int \ (b \ mod \ 2 \ ^ \ \text{len_of \ TYPE('a)}) = word_of_int \ b"}\]

The “sensible” definition of \texttt{word_of_int} has other convenient consequences. For example, when we define addition of words by \texttt{word_add_wi}, where \(u\) and \(v\) are words of the same length (and this definition does not involve the addition of \texttt{bins} which are not representatives of words), we also can prove the result \texttt{wi_hom_add} where \(a\) and \(b\) can be \textit{any} integers, whether or not they are values which represent words.

\[\text{\texttt{word_add_wi} : \\ \phantom{\text{\texttt{word_add_wi} : \}}\text{"u + v == word_of_int \ (uint \ u + uint \ v)"}}\]
\[\text{\texttt{wi_hom_add} = "word_of_int \ a + word_of_int \ b = word_of_int \ (a + b)"}\]

The following theorems, of the form \texttt{Rep(Abs \ x) = \ f \ x}, describe the behaviour of \texttt{Abs} outside the representing set \texttt{S}. (It follows that \texttt{range \ f = S}).

\[\text{\texttt{obin_int_obin} = "int_to_obin \ (onum_of \ n) = mk_norm_obin \ n"}\]
\[\text{\texttt{int_word_uint} = "uint \ (word_of_int \ a) = a \ mod \ 2 \ ^ \ \text{len_of \ TYPE('a)}"}\]
\[\text{\texttt{unat_of_nat} = "unat \ (of_nat \ (n::nat)) = n \ mod \ 2 \ ^ \ \text{len_of \ TYPE('a)}"}\]

We therefore defined an extended type definition predicate, as follows:

\[\text{\texttt{td_ext \ Rep \ Abs \ A \ norm} = \texttt{type_definition \ Rep \ Abs \ A \ & \ (ALL \ y. \ Rep \ (Abs \ y) = norm \ y)}\]

and we have extended type definition theorems including the following:

\[\text{\texttt{td_ext_int_obin} = "td_ext \ int_to_obin \ onum_of \ (Collect \ is_norm_obin \ mk_norm_obin"}\]
\[\text{\texttt{td_ext_uint} = "td_ext \ uint \ word_of_int \ (uints \ (len_of \ TYPE('a))) \ (bintrunc \ (len_of \ TYPE('a)))"}\]
\[\text{\texttt{td_ext_sbin} = "td_ext \ sint \ word_of_int \ (sints \ (len_of \ TYPE('a))) \ (sbintrunc \ (len_of \ TYPE('a) - 1))"}\]
\[\text{\texttt{td_ext_uint} = "td_ext \ uint \ word_of_int \ (uints \ (len_of \ TYPE('a))) \ (\%i. \ i \ mod \ 2 \ ^ \ \text{len_of \ TYPE('a))"}\]
\[\text{\texttt{td_ext_unat} = "td_ext \ unat \ of_nat \ (unats \ (len_of \ TYPE('a))) \ (\%i. \ i \ mod \ 2 \ ^ \ \text{len_of \ TYPE('a))"}\]

Since \texttt{Abs(Rep \ x) = x} it follows that \texttt{norm \ o \ norm = norm}, so we call it a normalisation function; we say \(x\) is normal if \(x = \text{norm} \ y\) for some \(y\), equivalently if \(x = \text{norm} \ x\). We also have \texttt{norm \ o \ Rep = Rep}, and \texttt{Abs \ o \ norm = Abs}. 
As we frequently had to transfer results about a function on one type to a corresponding function on another type we formalised some general relevant results. Consider a function \( f : \rho \to \rho \), where \( \rho \) is the representing type in a type definition theorem with normalisation function \( \text{norm} \). We say \( x \) and \( y \) are \( \text{norm-equiv} \) to mean \( \text{norm} \ x = \text{norm} \ y \). Then some or all of the following identities may hold:

\[
\begin{align*}
\text{norm} \circ f \circ \text{norm} &= \text{norm} \circ f \quad \text{f takes norm-equiv arguments to norm-equiv results} \\
\text{norm} \circ f \circ \text{norm} &= f \circ \text{norm} \quad \text{f takes normal arguments to normal results} \\
\text{norm} \circ f &= f \circ \text{norm} \quad \text{both of the above} \\
f \circ \text{norm} &= f \quad \text{f takes norm-equiv arguments to the same result} \\
\text{norm} \circ f &= f \quad \text{f takes every argument to a normal result}
\end{align*}
\]

Consider functions \( f : \rho \to \rho \) and function \( h : \alpha \to \alpha \), where \( \rho \) and \( \alpha \) are the representing and abstract types in a type definition theorem. These can be related in any of the following ways.

\[
\begin{align*}
h &= \text{Abs} \circ f \circ \text{Rep} & (1) \\
\text{Rep} \circ h &= f \circ \text{Rep} & (2) \\
h \circ \text{Abs} &= \text{Abs} \circ f & (3) \\
\text{Rep} \circ h \circ \text{Abs} &= f & (4)
\end{align*}
\]

Of these, (1) would be the typical way to define \( h \) in terms of \( f \), and (4) provides the most useful properties, as it implies all the rest; they all imply (1). As for the inverse implications, we obtained a number of general results showing when they are available, depending on which of the properties about \( \text{norm} \) and \( f \) above are satisfied (see [3, TdThs.thy]). For example, where \( \text{norm} \) is \( \text{bintrunc} \ n \), truncation of a \( \text{bin} \) to \( n \) bits, and \( f \) is addition (with two arguments), then \( f \) takes \( \text{norm-equiv} \) arguments to \( \text{norm-equiv} \) results. This is the key to obtaining the result \( \text{wi_hom_add} \) shown earlier, which is of the form of (3) above, from the definition \( \text{word_add_wi} \), of the form of (1). A similar situation applied in deriving \( \text{word_no_log_defs} \) (see §2.7).

Each type definition theorem is used by the functor \( \text{TdThms} \) or \( \text{TdExtThms} \) to generate a number of consequences, found in structures such as:

structure word =
  TdThms (struct val td_thm = type_definition_word ... end) ;
structure int_obin =
  TdExtThms (struct val td_ext_thm = td_ext_int_obin ... end) ;

We note in particular \( \text{word_nth.Rep_eqD} \) and \( \text{word_eqI} \), derived from it; \( \text{word_nth} \) selects the \( n \)th bit of a word, and is written infix as \( !! \).

\[
\begin{align*}
\text{word_nth.Rep_eqD} &= "\text{word_nth} x = \text{word_nth} y \implies x = y" \\
\text{word_eqI} &= "(!! n < \text{size} u \implies u !! n = v !! n) \implies u = v"
\end{align*}
\]

The latter was frequently useful in deriving equalities of words. For example, our function \( \text{word_cat} \) concatenates words. We had proved a theorem \( \text{word_nth_cat} \).
which gives an expression for \texttt{word\_cat} \( a \ b \ n \). Using results like these we could prove two words equal by starting with \texttt{word\_eqI}, and simplifying. This approach was often useful for proving identities involving concatenating, splitting, rotating or shifting words.

In the same way, the theorem \texttt{bin\_nth\_lem} was useful for proving equality of bins, where \texttt{bin\_nth} \( x \ n \) is bit \( n \) of \( x \), using theorems such as \texttt{nth\_bintr}.

\begin{verbatim}
bin\_nth\_lem = "bin\_nth x = bin\_nth y ==> x = y"
nth\_bintr = "bin\_nth (bintrunc m w) n = (n < m & bin\_nth w n)"
\end{verbatim}

\section*{2.7 Simplications, number\_of, literal numbers}

As noted earlier, the type \texttt{bin} is used in connexion with the function \texttt{number\_of} :: \( \text{bin} \Rightarrow \text{number} \) to express literal numbers. When a number (say 5) is entered, it is syntax-translated to \texttt{number\_of} (\texttt{Pls BIT B1 BIT B0 BIT B1}). The function \texttt{number\_of} is defined variously for various types and classes, e.g.:

\begin{verbatim}
int\_number\_of\_alt = "number\_of (w::int) :: int == w"
word\_number\_of\_def = "number\_of (w::bin) :: 'a::len0 word == word\_of\_int w"
\end{verbatim}

\textbf{Simplifications for arithmetic expressions} Certain arithmetic equalities, such as associativity and commutativity of addition and multiplication, and distributivity of multiplication over addition, hold for words. We wrote an function \texttt{int2lenw} in Standard ML to generate a number of results for words, in \texttt{word\_arith\_eqs}, from the corresponding results about integers. See the file \texttt{WordArith.thy} for details. From these and other results, we showed that the word type is in many of Isabelle's arithmetical type classes (see \texttt{WordClasses.thy}). Therefore many automatic simplifications for these type classes are available for the word type. Thus, for example

\[ a + b + c = (b + d :: 'a :: len0 word) \]

\texttt{Isabelle} is set up to simplify arithmetic expressions involving literal numbers as \texttt{bins} very effectively, using simplification rules which in effect do binary arithmetic, provided that the type of the numbers is in the class \texttt{number\_ring}. This is the case for words of positive length; unfortunately this does not work for zero-length words, since Isabelle's \texttt{number\_ring} class requires \( 0 \neq 1 \). Thus an expression such as \( (6 + 5 :: 'a :: len word) \) gets simplified to \( 11 \) automatically, regardless of the word length, which need not be known. Another standard simplification involves the predicate \texttt{iszero}, so \( (6 + 5 :: 'a :: len word) = 7 \) gets simplified to \( \texttt{iszero} (4 :: 'a :: len word) \).

Further simplification of such expressions, i.e., from \( (11 :: \texttt{word2}) \) to \( 3 \) (where \texttt{word2} is a type of words of length 2) and from \texttt{iszero} \( (4 :: \texttt{word2}) \) to \texttt{True} depend on the specific word length. We would want to use a theorem like \texttt{num\_of\_bintr}, but we cannot reverse it to use it as a simplification rule because it would loop. Instead we can simplify using \texttt{num\_abs\_bintr} (which is derived from \texttt{num\_of\_bintr} and \texttt{word\_number\_of\_def}).
num_of_bintr =
"number_of (bintrunc (len_of TYPE('a)) (b::bin)) = number_of b"

num_abs_bintr =
"number_of (b::bin) = word_of_int (len_of TYPE('a)) b"

We then need to simplify the word length definition, using the theorem giving
len_of TYPE('a) for the specific type, then simplify using bintrunc_pred_simps,
which simplifies an expression like bintrunc (number_of bin) (w BIT b), and
finally apply word_number_of_def in the opposite direction.

Given an expression such as iszero (4 :: word2), we can use the theorem
iszero_word_no as a simplification rule, and it doesn’t loop because the type of
number_of ... (the argument of iszero ( )) is a word on the left-hand side
but is an int on the right-hand side. We would then simplify using the rule
giving the word length and bintrunc_pred_simps.

iszero_word_no = "iszero (number_of (bin::bin)) =
    iszero (number_of (bintrunc (len_of TYPE('a)) bin))"

A further approach to simplifying a literal word is to simplify an expression
such as uint (11 :: word2), which means converting (11 :: word2) to
the integer in the range uints 2, i.e. 0...2^n – 1. We would simplify using
uint_bintrunc, the rule giving the word length and bintrunc_pred_simps.

uint_bintrunc = "uint (number_of (bin::bin)) =
    number_of (bintrunc (len_of TYPE('a)) bin)"

Note that in uint_bintrunc the two instances of number_of have result types
word and int respectively. Corresponding theorems are available for the signed
interpretation of a word, and to simplify unat of a literal.

Simplifications for logical expressions These are more difficult because we
do not have a built-in type class. The definition of the bit-wise operations, and
how from the definitions we obtained simplifications such as bin_not_simps and
bin_and_Bits, is described in §2.3.

A literal expression such as 22 & 11 can be simplified first using the (de-
derived) rules word_no_log_defs (the actual definitions being word_log_defs)

word_log_defs = ["u & v ==
    number_of (bin_and (uint u) (uint v))", ...
word_no_log_defs = ["number_of a & number_of b ==
    number_of (bin_and a b)", ...

and then using the simplifications such as bin_and_Bits (word_no_log_defs
and many rules for bit-wise logical operations on bins are in the default simpset).

We derived counterparts for bins of commonplace logical identities such as
associativity and commutativity of conjunction and disjunction, and others such
as (x & y) v x = x. We wrote Standard ML code to use these to generate coun-
terparts of these for words, so that one function, bin2lenw, sufficed to generate
all the corresponding results, found in word_bw_simps, about logical bit-wise
operations on words. See the file [3, WordBitwise.thy] for details.
Simplifications for literals We wanted to have automatic simplifications for literal expressions in the default simpset. But to avoid using these where they were not wanted, we often had to install only a special case of a theorem (generally, where some (sub-)expression is of the form `number_of x`) as a default simplification rule. In other cases we needed to use simplification procedures, which may apply a simplification rule or not, depending on the form of a term.

Special-purpose simplification tactics Consider the result (for words) "(x < x - z) = (x < z)"; each inequality holds iff calculating x - z causes underflow. Several results required about words, such as this one, could be proved by translating into goals involving sums or differences of integers, together with case analyses as to whether overflow or underflow occurred or not. So we developed tactics for these: `uint_pm_tac` does the following

- unfolds definitions of ≤, using `word_le_def` (similarly for <)
- unfolds occurrences of `uint (a + b)` using `uint_plus_if`
  (similarly for `uint (a - b)`)
- for every occurrence of `uint w` in the goal, inserts `uint_range`
- solves using `arith_tac`, an Isabelle tactic for solving linear arithmetic

`word_le_def = "a <= b == uint a <= uint b"
uint_plus_if' = "uint (a + b) = (if uint a + uint b < 2 ^ len_of TYPE('a) then uint a + uint b else uint a + uint b - 2 ^ len_of TYPE('a))"
uint_range' = "0 <= uint w & uint w < 2 ^ len_of TYPE('a)"

This proved effective for a reasonable number of goals that arose in practice; it relies on the fact that `arith_tac` is very effective for goals involving <, <-, + and – for integers. Details of the code are in [3, WordArith.thy].

A similar method was used to solve a problem posed by a referee: to prove that, in signed n-bit arithmetic, adding x + y overflows, i.e., `sint x + sint y ≠ sint (x + y)`, iff `(((x+y)^x) & ((x+y)^y)) >> (n - 1)` (in C) is non-zero.

2.8 Types containing information about word length

We have defined types which contain information about the length of words. For example, `len_of TYPE(tb t1 t0 t1 t1 t1) = 23` because `t1 t0 t1 t1 t1` translates to the binary number 10111, that is, 23. The relevant simplification rules (which are axioms, and so in the default simpset) are

`len_tb : "len_of TYPE (tb) = 0"
len_t0 : "len_of TYPE ('a :: len t0) = 2 * len_of TYPE ('a)"
len_t1 : "len_of TYPE ('a :: len0 t1) = 2 * len_of TYPE ('a) + 1"

and so `len_of TYPE(tb t1 t0 t1 t1 t1)` is simplified to 23 automatically.

We use the type class mechanism to prevent use of the type `tb t0` (corresponding to a binary number with a redundant leading zero); the class `len` is used for words whose length is non-zero and we used the arity declarations shown, although the instance declarations shown are then deducible.
arity tb :: len0
arity t0 :: (len) len0 instance t0 :: (len) len
arity t1 :: (len0) len0 instance t1 :: (len0) len

By the arities declaration for t0, we can make use of a type α t0 only where α is in the class len (indicating a non-zero word length), which prevents using tb as α. The deduced instance results mean that any type α t1 is of class len, and likewise for α t0, when α is of class len.

It is also possible to specify the word length rather than the type, and have the type generated automatically. For example, for a goal with a variable type, e.g. "len_of TYPE(?a :: len0) = 23", repeated use of certain introduction rules (len_no_intros) will instantiate the variable type ?a to tb t1 t0 t1 t1 t1.

See [3, Autotypes.thy] for details, and for further relevant theorems. Brian Huffman of Galois Connections has developed types in a similar way, and syntax translation so that the length can be entered or printed out as part of the type.

2.9 Length-dependent exhaust theorems

Consider the goal "((x :: word6) >> 2) || (y >> 2) = (x || y) >> 2" where x >> 2 means x, with bits shifted two places to the left, and x || y is bit-wise disjunction. While this is an example of a general theorem which might well have been provided in the development of the theories, there would be a large number of such theorems, not all of which have been provided.

We could prove such a theorem by expanding x by

\[ x = \text{Pls BIT xa BIT xb BIT xc BIT xd BIT xe BIT xf} \]

(similarly y) and calculating both sides by simplification. To enable this we generate a theorem for each word length; the one for word length 6 is shown.

"[| !!b ba bb bc bd be. w = number_of (Pls BIT b BIT ba BIT bb BIT bc BIT bd BIT be) ==> P; size w = 6 ||] ==> P"

We also generated theorems to express a word as a list of bits; for example, for x of length 6, expressing to_bl x as [xf, xe, xd, xc, xb, xa].

Such a theorem can then be instantiated; for example, for the goal above, one would use the theorem for word length 6 twice, instantiating it with x and y respectively. An example is in [3, Word32.ML].

We are also developing techniques for translating a goal into a format suitable for handing over to a SAT solver. This involves expressing a word of length n as a sequence of n bits, and we have used these theorems for this purpose also.

3 Conclusion

The theories we describe have been used extensively in the NICTA’s L4.verified project, which requires reasoning about the properties of machine words and
their operations. We have discussed how we defined types of words of various lengths, with theorems which apply to words of any length. We have shown how to make definitions about bins by a procedure sufficiently resembling primitive recursion to be practical and useful. We have taken advantage of the fact that the set of words is isomorphic to several different sets and used “pseudo” type definition theorems to use these and derive relevant results in an efficient and uniform way. Finally we described other useful techniques, such as how to create types which automatically imply the word length, using type constructors corresponding to binary digits.

In these theories, where a single type of words has a definite length, definitions and theorems relating to concatenating or splitting words were difficult. In this aspect, the use of PVS, with its more powerful type system, and its bit-vector library [2], might be easier.

A noteworthy feature of the work was the value of Standard ML as the user interface language. As described in §2.6 we used its structures and functors, which were very convenient for generating a large number of theorems of the same pattern without repeating code. We used its capabilities as a programming language to write a number of functions for generating theorems en masse, such as the SML function int2lenw and bin2lenw which were used to generate respectively 15 and 31 theorems about words from corresponding theorems about ints and bins. Coding in SML was also indispensable for the simplification procedures used to provide automatic simplification of literal expressions, for tactics such as uint_pm_tac, for generating the theorems of §2.9 for arbitrary n, and for HOL-style conversions, which were occasionally used in the proofs. Of course, more mundane uses of its capabilities, such as applying a transformation to a list of theorems, was commonplace in our work.

Acknowledgements I thank Gerwin Klein and anonymous referees for helpful suggestions, and John Matthews for the contribution of Galois Connections.

References

An Architecture for Extensible
Click’n Prove Interfaces

Holger Gast
Wilhelm-Schickard-Institut für Informatik
Sand 13, D-72076 Tübingen
University of Tübingen

Abstract. We present a novel software architecture for graphical inter-
faces to interactive theorem provers. It provides click’n prove func-
tionality at the interface level without requiring support from the un-
derlying theorem prover and enables users to extend that functionality
through light-weight plugins. Building on established architectural and
design patterns for interactive and extensible systems, the architecture
also clarifies the relationship between the special application of theorem
proving and conventional designs.

1 Introduction

Click’n prove [1], or prove-by-pointing [7], user interfaces for interactive theo-
rem provers enable experts to edit proof scripts more efficiently and make the
theorem prover more accessible to beginners. They visualize the proof state [28,
17] and interpret mouse gestures as commands that exhibit specific subterms of
a goal [7, 9], perform context-sensitive rule applications [1, 6], or manipulate the
proof script in a structure-oriented manner [9, 6]. Similar techniques have proven
successful in software verification systems [17, 1, 22].

In previous designs, the click’n prove interface requires substantial support
from the theorem prover: to interpret a mouse click as term selection, they
assume that the prover augments its output with markups of the term structure
[30, 8, 9, 21, 28, 3, 5]. Alternatively, the responsibility for pretty-printing terms is
transferred to the interface entirely, and the prover delivers an encoding of its
internal data structures [28, 30, 9, 6]. Some proposals [21, 3] extend the prover
itself to handle click’n prove actions. While these approaches thus reuse the
functionality available in the prover, they also require the prover to be modified
for each specific interface and the invested effort cannot be transferred easily to
other interfaces or provers.

In this paper, we propose an architecture for click’n prove interfaces in which
the prover does not have to be aware of the interface and in particular does
not have to be modified. The approach thus rests on a fundamental design rule
for interactive applications [11, §2.4]: business (or application) logic and pre-
sentation logic should be strictly separated. While the business logic, i.e. the
prover, remains stable over many releases, the user interface is subject to fre-
quent changes that accommodate the needs of different user groups or allow the
system to run on different platforms. We use a general context-free, incremental parsing algorithm at the interface level to expose the subterm structure without breaking this design rule.

Our main objective is extensibility: previous designs required programming at the prover level to accommodate new click’n prove actions [8, 21] or were restricted to configuring a generic mechanism [29, 9, 3]. Instead, we propose that users should provide the functionality necessary for their daily work through light-weight plugins. The construction of these plugins must not require any knowledge about the theorem prover’s implementation, and only modest insights into the architecture of the graphical interface. The general approach imitates the Eclipse plugin model [15], but reduces its complexity by retaining only those aspects that are immediately necessary.

The current implementation of the architecture realizes the given objectives for the Isabelle [25] theorem prover. The example plugins presented (Section 4) demonstrate that little effort will be necessary to provide similar functionality for different provers.

Structure of the Paper Section 2 motivates and describes the proposed architecture. Section 3 treats our parsing algorithm in more detail. Section 4 gives example plugins that provide click’n prove support for Isabelle/HOL [24]. Section 5 compares the proposal with related work. Section 6 outlines future directions and discusses design decisions. Section 7 concludes.

2 Architecture

To define our architecture, we first enumerate the essential design forces. We proceed with the division of responsibilities between the interface and the prover, and describe the extensibility mechanism. Finally, we discuss the steps necessary...
to apply the architecture to specific theorem provers. The presentation stresses
the standard architectural and design patterns for graphical user interfaces and
extensible systems (e.g. [11, 26]) underlying our solution. In this way we hope
to offer a new perspective on the commonalities and differences in software de-
sign between graphical interfaces for theorem provers and more conventional
applications.

2.1 Design forces

The following considerations govern the decisions taken in the architecture.

Changing requirements User interfaces must often be changed to add new
functionality for different user groups. In particular, users of theorem provers
profit most from click’n prove support if it addresses the situations they
encounter most frequently. We thus envision that theories and corresponding
support will be developed in parallel [29, 1].

User-level extension We expect that users will implement the support they
require directly, rather than wait for the system developers to provide it.

Disjoint developer groups The theorem proving system consists of three,
largely independent parts: the theorem prover, a core framework for the
graphical user interface, and theory-specific click’n prove actions. We as-
sume that these parts will be developed and maintained by three, largely
disjoint groups of programmers who are familiar with their own code only.

Independent extension We expect that contributions to click’n prove func-
tionality are most useful if they can be combined into a consistent environ-
ment in a flexible manner.

Complementary requirements The data structures of provers are optimized
for the operations occurring in proof search and proof checking, but do not
necessarily offer the operations required by interactive interfaces. Two ex-
amples for this are the navigation through a tree structure from nodes to
their parents, and error recovery in parsing.

2.2 Separation of Application Logic and Presentation

Figure 1 exhibits the main components of our architecture and the connections
between them. We now describe the left part of the figure, leaving the plugin
management to Section 2.3. The system is divided into three layers: the prover
layer, the model layer and the display layer.

The prover layer encapsulates the basic communication with the theorem
prover. We assume that the prover provides a read-eval-print loop, which reads
one textual command at a time from standard input, executes it and sends some
textual result to the standard output. To accommodate differences in protocol,
such as the prompt of the loop or the terminator for commands, a prover-specific
configuration component is invoked in every communication. The prover layer
provides a service to execute a single command and notifies the higher levels by
an Asynchronous Completion Token [26, §3] when the answer has been received.

The model layer manages the data structures that are eventually displayed on the screen as dictated by the Model-View-Controller (MVC) pattern [11, §2.4]. The components proof script and proof state store the textual representation of the current script and state as sent to and received from the prover. The management of the proof script is standard [9, 5, 2]. Both the script and the state are observed by (see [16]) incremental chart parsers (see Section 3) that maintain parse trees (or parse DAGs, in case of ambiguities) of the content. The interactions between the proof script, the prover component, and the user interface are complex. For example, when a command is sent, it must be contributed to the locked region of the script; if it fails or is undone later, the lock must be removed. This situation suggests a Mediator [16] (see also [4]), which is responsible for managing the collaboration between the connected components.

The display layer is divided broadly into view and controller components, as suggested by the MVC pattern. The view components may, however, incorporate some controller functionality for entering the proof script by keyboard, and therefore implement the Document-View variant [11]. Besides the textual representation of their model data, they also access the chart for syntax highlighting, for instance to distinguish free and bound identifiers.

Gesture recognition is the central component for click’n prove functionality: it registers for mouse events, including drag&drop events, on all views and transforms them into click-and-prove events that it forwards, through the dispatcher component, to registered click’n prove actions. With these responsibilities, gesture recognition is a typical controller [11, §2.4], which receives raw, system-dependent events and interprets them as high-level, application-specific events. Gesture recognition also implements term selection: when the user clicks to one of the views, it uses the selected character to ask the chart for the selected token. To accommodate ambiguous grammars, it then identifies a selected path as the longest path through the parse tree(s) upwards from the selected token. Finally, the selected tree is the smallest, i.e. the lowest, tree on the selected path. This protocol allows the user to select a specific tree by pointing to a token that is contained in none of its subtrees, for instance by pointing to the operator symbol of an expression.

### 2.3 Extensibility

We use the INTERCEPTOR pattern [26, §2] to achieve extensibility. In that pattern, a framework defines interception points for which extensions, or interceptors, can register to provide new services. The framework’s behavior is specified by a finite automaton, whose transitions directly correspond to the available interception points. When the automaton takes a particular transition, a descriptive event is sent to the registered interceptors via a dispatcher component. The interceptors are then given the opportunity to query the framework’s state and modify its future behavior through a context object. The context object, as a FACADE object [16], accesses all of the components in the model layer, which
The dashed arrows indicate the concrete events of our architecture will be defined in Section 2.4.

The main benefit of the Interceptor approach to extensibility is that extensions are light-weight objects, which have to implement only a restricted interface to receive events from the framework. Their developers need to be familiar only with the framework’s specification as a finite automaton and with the context object; the internal realization remains hidden. The experience with the example plugins (Section 4) suggests that users who are moderately familiar with the Java programming language will be able to contribute extensions to suit their particular needs.

Our architecture complements this setup with a loader component, which scans a plugins directory on startup and registers all found extensions (see [13, 15]). We implement lazy evaluation [15]: the plugin declares interceptors in an XML document plugin.xml, which contains sufficient information to provide the user interface representation. The implementing classes are loaded only when the user triggers an action.

2.4 Events

Currently, the framework defines two events: the TreeSelectEvent occurs when the user selects a subterm with the mouse and requests a menu with the applicable actions. The TreeDefaultSelectEvent occurs when the user double-clicks on a subterm. Both events are characterized by a location, and the selected token, tree, and path as defined in Section 2.2. The location designates the view in which the event occurred, and is currently either script or state.

Dispatching of tree events proceeds in two steps: first, all actions registered for the event are queried whether they are enabled for the particular event. That condition is given by a boolean predicate on the event’s parameters (see Section 4). Second, one action to be invoked is selected. In the case of a TreeSelectEvent event, the user chooses from a popup menu. For the TreeDefaultSelectEvent event, the dispatcher checks whether there is exactly one enabled action and if so, it invokes that action.

2.5 Specialization for a Theorem Prover

To apply the framework to a given theorem prover, it is necessary to write the configuration component for the communication protocol, to produce grammars for the prover’s input language and the proof state display, and to program plugins that react to mouse gestures. Since the plugins will necessarily refer to specific non-terminals and node labels in the syntax trees and will furthermore generate prover-specific commands, they cannot be reused for different provers.

In the case of Isabelle, the required grammars can be obtained in a straightforward fashion: Isabelle supports several logics, which build, however, on a meta-logic Pure, whose syntax for terms and propositions is given on a single page in [25]. The remaining productions are declared explicitly in the theories defining the various logics, and can be extract using only the Pure grammar.
The syntax for commands, however, is given by (backtracking) parsing functions, such that its grammar must be provided by hand. The task has been finished within two days, using on the existing reference manuals for the input and code inspection for the output.

3 Parsing the Proof Script and State

The interpretation of mouse gestures requires a suitable internal representation of the material displayed on the screen. Since the proof state and the proof script are given as text documents, a straightforward solution is to construct a parse tree (or a parse DAG, in case of ambiguities). To support the Isar notion of proof scripts as self-contained documents [23], click’n prove actions must be applicable to the script as well as the proof state. Parsing at the interface level is a major requirement of the proposed architecture, and we describe our algorithm in some detail to demonstrate that the approach is viable. The Isar input language and the proof state display of Isabelle pose three major challenges:

1. They contain nested languages, with an outer syntax that describes the overall structure and an inner syntax for terms and propositions.
2. Both languages are extensible by declarations in theories and require full context-free grammars.
3. The proper handling of the proof script requires incremental parsing.

Before giving our solution, we review briefly why existing, mainstream approaches to parsing fail to meet these challenges.

3.1 Existing Approaches

Parsing with nested languages has been studied extensively. The accepted solution is to incorporate the lexical analysis into the context-free grammar. The ASF+SDF tool [32] handles full context-free grammars using GLR parsing [31], but adding the lexical analysis results in extremely large LR-automata that preclude on-the-fly generation for extensible grammars. The Harmonia framework [18] also uses GLR parsing. Its proposed solution to nested languages is to associate scanners with non-terminals, but this extension has not been implemented so far. It is also unclear how extensible grammars could be supported. PackRat (or PEG) parsing [14] uses a backtracking recursive descent parser with memoization and integrates lexical and syntactical analysis. Since parser generation is a simple process, extensible grammars could be implemented. However, PackRat parsing only works for deterministic grammars without left-recursion.

Approaches based on structured editing (e.g. [10]), which have been used for theorem provers before [30, 9], seem to deviate too much from the usage of modern IDEs: users expect parsing to proceed in the background without restricting the possible edits, and to finish shortly after they have produced syntactically correct input.
The ProofGeneral project [5] and CtCoq [6] assign the responsibility for parsing commands to the theorem prover. Unfortunately, the current version Isabelle 2005 does not produce parse trees for commands, but encapsulates commands as transitions, which are (ML) functions from states to states. It thus appears that to obtain more detailed information, the parser for the Isar outer syntax would have to be rewritten almost entirely.

3.2 Incremental Chart Parsing

Chart parsing [20] works with general context-free grammars and can handle incremental parsing [33, 27]. Since it does not require a generation phase, it is a light-weight solution to extensible languages. Perhaps the best-known chart parser is Earley’s algorithm [12], which Isabelle uses for its inner syntax.

To define chart parsing, let $G = (N, T, P, S)$ be a grammar with non-terminals $N$, terminals (or tokens) $T$, productions $P$, and a start symbol $S$. We assume that the right-hand side of a production either contains only non-terminals or consists of a single terminal symbol, in which case it is called a pre-terminal production. Let $s$ be an input string that lexical analysis has split into tokens $t_1 \ldots t_n$. The chart is a directed graph $(V, E)$ where the vertices $V = \{0, \ldots n\}$ mark the positions between the tokens, before the first, and after the last token. The edges $E$ are triples $(v, v', A \rightarrow \alpha A' \cdot \alpha')$ over vertices $v$, $v'$ and an item, such that $A \rightarrow \alpha A'$ is a production in $P$. The dot indicates the current position in the parsing process. An edge is active if $\alpha' \neq \varepsilon$, and inactive otherwise.

The chart is initialized by adding a pre-terminal edge $e_i = (i - 1, i, T_i \rightarrow t_i)$ for each token $t_i$ and an edge $(0, 0, S \rightarrow \cdot \alpha)$ for each production $S \rightarrow \alpha \in P$. Then, the following steps are applied until no new edges are produced:

**predict** For each edge $(v, v', A \rightarrow \alpha \cdot B \alpha')$ in the chart and each production $B \rightarrow \beta \in P$, add an edge $(v', v', B \rightarrow \cdot \beta)$.

**combine** For each active edge $(v, v', A \rightarrow \alpha \cdot B \alpha')$ and inactive edge $(v', v'', B \rightarrow \beta \cdot \cdot \cdot)$ in the chart, add an edge $(v, v'', A \rightarrow \alpha B \cdot \alpha')$

There are only finitely many possible edges and the process terminates with a chart that contains an edge $(v, v', A \rightarrow \alpha \cdot)$ iff there is a derivation $A \Rightarrow \alpha \Rightarrow^* t_{v+1} \ldots t_v$. To enumerate all parse trees, it is sufficient to mark edges with a unique identifiers, and to modify the combine step to record in the result the identifier of the referenced, inactive edge. We say that an edge $e$ depends on an edge $e'$ [33] iff $e$ references $e'$, or $e$ depends on some edge $e''$ that depends on $e'$. It is straightforward to extend the framework to priority grammars [25, §7.1], such that associativity and precedence of operators can be encoded.

Wirén’s algorithm for incremental parsing [33] takes a chart and a single modification that consists in the addition, deletion, or replacement of some token $t$. Multiple changes are processed in order. The algorithm splits the chart at the point of modification by removing all edges spanning the modification and renumbering the vertices to accommodate the insertion or deletion of $t$. After adding the pre-terminal edge for a newly inserted token, the normal parsing algorithm is run to complete the chart for the modified input.
Our algorithm improves on Wirén’s in that edges reference equivalence classes of edges, where two edges are equivalent if they have the same start and end vertices and their items have the same left-hand side. With this definition, the algorithm generates a packed representation of the parse forest that results from ambiguous grammars. This packed representation also allows for more efficient incremental parsing by lazy edge removal: when some edge $e$ becomes invalid due to a modification, we run the chart parser locally to produce a new edge $e'$ that is equivalent to $e$. If such an edge is found, the dependent edges of $e$ do not have to be removed at all, and parsing finishes.

To accommodate nested languages, we introduce lexer switching. A switch to lexer $l$ is marked by $↑l$ in the right-hand side of productions. The chart structure is generalized such that a token $t$ can be surrounded by arbitrary vertices $v$, $v'$, which are not necessarily numbered consecutively. When the chart parser encounters an edge $(v, v', A \rightarrow α \cdot ↑l α')$, it searches for a token $t$ that is located in the input string at the end of token $v'$ and has been produced by lexer $l$. If $v''$ is the vertex immediately before $t$, the algorithm creates an edge $(v, v'', A \rightarrow α ↑l α')$, such that parsing proceeds with the pre-terminal edge of $t$. For efficiency, tokens are generated lazily when some production requests them.

4 Plugins

This section demonstrates how plugins can implement click’n prove functionality based on the framework proposed in Sections 2 and 3. We use the current implementation which is developed in Java using the SWT and JFace libraries [13] for the user interface. The necessary grammars (Section 2.5) have been provided by hand. The examples are chosen to illustrate the services available from the framework, but do not provide a comprehensive click’n prove interface for Isabelle. Since the framework API is still under development, we cannot yet present a complete specification. We include the actual code of the plugins to substantiate the claim that only modest experience is necessary to produce them.
4.1 Simplification

Isabelle’s simplifier is used in tactic-style proofs to rewrite the first goal of the current state with a set of equality theorems declared as simplifier rules. In the click’n prove interface it should therefore be available in the context menu when the user points to the first goal of the proof state.

Figure 2(a) shows a screenshot: the left-hand side panel contains the proof script in which the locked region is marked by a red background. The right-hand side is split between the proof state on top and a panel for error messages. By pointing the mouse to the \longrightarrow operator, the user has selected the goal term as indicated by the highlight. The term is also shown in prefix notation in the status line. A click on the right mouse button has brought up the action menu, where the simplifier action is available.

When the user selects the menu entry, the run() method of the following delegate class is invoked. The class implements the TreeSelectActionDelegate interface associated with the TreeSelectEvent (Section 2.4). The run() method uses the context object to insert a new command to the proof script.

```java
public class Simplify implements TreeSelectActionDelegate {
    public void run(ClickProveContext ctx, TreeSelectEvent ev) {
        ctx.insertAndSubmitGeneratedCommand("apply (simp)";
    }
}
```

To make the simplifier action appear in the context menu, the following XML element is included in the plugin descriptor (Section 2.3). It specifies the internal identifier of the action, the label for the menu entry and the delegate class containing the implementation. Furthermore, the element declares the enable condition for the action: the location of the tree select event must be the proof state and the selected path must include a subgoal node, which is directly below a subgoals node representing the list of all subgoals, which again is directly below the ps_goals node, which captures the lower part of Isabelle’s proof state. Paths are read from the bottom to the top, paralleling the nesting of parse trees. The XML fragment thus expresses the condition that the user has selected some term within in the first goal of the proof state.

```xml
<tree-event id="tactic.simplify" label="simplify" class="tactic.Simplify">
    <enable>
        <and>
            <location id="state"/>
            <path><node nt="ps_goals"/>
                <node nt="subgoals" label="cons"/>
                <node nt="subgoal"/>
                <any/>
            </path>
        </and>
    </enable>
</tree-event>
```
The conditions on the selected path expressible in XML are modeled after regular expressions: \texttt{node} requires a node with a specific label and/or a non-terminal; the attribute \texttt{pos="i"} indicates that the node must be the \textit{i}th child of the next node on the selected path. The condition \texttt{any} matches an arbitrary sequence of nodes; \texttt{alt}, \texttt{maybe}, \texttt{repeat}, and \texttt{seq} represent the regular operators for alternative, option, Kleene star, and sequence.

4.2 Rule application

Our second example is an action that searches for the rules applicable to the conclusion of the first goal. It demonstrates that with more collaboration from the prover, more sophisticated click’n prove actions become possible. Isabelle 2005 provides a command \texttt{find\_theorems}, which searches the current proof context for theorems fulfilling a given condition. The condition \texttt{intro} finds rules that are applicable to the first goal. Figure 2(b) shows the resulting screenshot: the user has pointed to the first goal and selected from the context menu the action \texttt{apply rule}, which has queried the theorem database and opened the dialog.

The \textit{enable} condition for the new action is similar to that for simplification. The following code sequence is taken from the delegate's \texttt{run()} method and demonstrates how commands that do not belong to the proof script can be sent to the prover.

```java
TheoremSelectDialog sel = new TheoremSelectDialog(ctx.getShell());
Command c = new TextCommand("find\_theorems intro");
ActiveCommand cmd = ctx.submitSilentCommand(c);
try {
    cmd.addListener(new ReportFound(ctx, sel));
    if (sel.open() == Window.OK) {
        String thm = sel.getSelectedTheorem();
        if (thm != null) {
            ctx.insertAndSubmitGeneratedCommand("apply (rule \"+thm+\")");
        }
    }
}
```

The \texttt{ActiveCommand} object implements the \textsc{Asynchronous Completion Token} pattern [26]: the context submits the given \texttt{find\_theorems} command to the prover for execution, but the call to \texttt{ctx.submitSilentCommand} returns immediately.\footnote{This behavior is essential because the \texttt{run()} method is invoked within the GUI event-thread and no new events are accepted before it returns – the user interface is "frozen". The call to \texttt{sel.open()} does not return immediately either, because the dialog is modal, but this situation is handled internally by the SWT library.} The \texttt{ReportFound} object is registered as an observer that will be notified when the answer from the prover has arrived. It scans the output and inserts the found theorems into the open \texttt{TheoremSelectDialog}. When the user selects one of those theorems, it is applied to the current state.
4.3 Quick Introduction and Elimination

The search for applicable rules using the action from Section 4.2 may take a few seconds, which is unacceptable when the desired rule is “obviously” the standard introduction or elimination rule. The QuickIntro and QuickElim classes therefore hold an extensible mapping from non-terminal/label pairs to the standard rules. For the Isabelle/HOL logic, for example, QuickElim maps (logic, exists) to exE and (logic, and) to conjE. The following code fragment is registered for the TreeDefaultSelectEvent, i.e. a double-click on one of the premises of the first goal.

```java
Object nt = ev.getTree().getNonTerminalID();
String l = ev.getTree().getLabel();
String rl = rules.get(new NTLabelPair(nt, l));
if (rl == null) {
    ctx.showErrorDialog("No quick elim rule for " + nt + ";" + l);
} else {
    ctx.insertAndSubmitGeneratedCommand("apply (erule+" + rl + ")");
}
```

Since the QuickIntro is defined analogously, we must give a precise enable condition for QuickElim to ensure mutual exclusion. QuickElim is applicable only under the first subgoal’s top-level meta-implication, and here to any one of the list of premises or the single premise, if there is just one.

```xml
<path>
  <node nt="subgoal"/>
  <maybe><node nt="logic" label="meta-all"></maybe>
  <node label="meta-implies"/>
  <alt>
    <seq><node pos="0"/>
      <repeat><node nt="bigimp_prems"/></repeat>
      <node/>
    </seq>
    <node pos="0"/>
  </alt>
</path>
```

4.4 Picking a fact in Isar

The following action demonstrates the use of tree navigators to access the parse DAG, starting from the selected tree. Bertot [6] shows that a similar access to the syntax tree is sufficient to implement proof-by-pointing [7].

The grammar for Isar proof scripts [23] includes named assumptions and intermediate facts of the form id: "proposition". To use a fact in a proof, it must be “picked” with the command from. The following simple action, registered for TreeDefaultSelectEvent, enables the user to generate the command by double-clicking anywhere within named fact.
A TreeNodeNavigator object represents a set of positions reached in the current navigation. The `navigate()` method of a tree node returns the tree node itself. The navigator’s `up()` method searches the ancestor nodes for a non-terminal/label combination and returns a navigator for these. The next steps navigate down to the first child, check that a name is present (it is optional in the grammar), and finally access the name. Since grammars can be ambiguous, tree navigation in general yields several results and the TreeNodeNavigator class encapsulates the backtracking search. The user can retrieve the results by the statement

```java
for (TreeNode p: navigator)
```

Alternatively, the method `single()` returns a single result if it exists or throws an exception.

The proof script is parsed incrementally, such that more sophisticated versions of the Pick action could examine – through the context object – the preceding commands to determine the exact command to be generated: after a `from` command, the prover is in chain mode and does not accept a second `from`. Instead, the first `from` must be augmented with a new `and` clause. After a goal statement, i.e. in show mode, a using command should be issued.

5 Related Work

User interfaces for theorem provers have attracted the attention of numerous researchers. In the following, we therefore focus on the main points of comparison: the collaboration with the prover that enables click’n prove functionality, and the mechanisms for extensibility. The existing solutions for parsing the proof script have been discussed in Section 3.1.

Bertot, Théry, and Kahn [30, 7, 9, 6] have introduced and developed the proof-by-pointing paradigm. They expect the prover to send output as trees, which they render on the screen using the PPML formalism of CENTAUR [10]. The translation of mouse gestures to proof commands takes place at the interface level [6]; the editor for the proof script is structure-oriented, requiring the user to manipulate abstract syntax trees, but also providing extensive support [6].

A similar implementation strategy is used in [8, 21, 3], where the prover marks each subterm in the output with its path $p$ in the overall syntax tree. Bertot et al. [8] interpret a mouse click by sending a command `pbp p` (for proof-by-pointing) to the prover, which generates a command, executes it, and sends it back to the interface for inclusion in the proof script. Aspinall and Lüth [3] generate commands from templates at the interface level.

The Jape editor for proof documents [29] includes a theorem prover that appears to share its data structures with the interface, such that mouse gestures can be readily interpreted. In the KIV [17] and Jive [22] verification systems, the graphical front-end is likewise coupled tightly to the built-in prover. The
interface for the distributed system $\Omega$mega [28] expects the prover to transmit its internal data structures to enable the elaborate and fine-grained visualization.

Extensibility has been achieved by two means: generic algorithms that can be configured for specific application domains, and broker architectures that provide a common infrastructure to which provers and front-ends can attach to exchange messages.

Bertot and Théry [9] generalize the proof-by-pointing algorithm to accommodate new connectives. The implementation in [8] likewise uses an extensible repository of functions that handle specific connectives, but they must be programmed at the prover level.

The Jape [29] editor can be configured by the inference rules of concrete logics. The declaration of a rule also includes the gesture that is to invoke the rule. The set of available gestures is large, but fixed, such that, for example, specific dialogs cannot be programmed.

Lüth and Wolff [21] introduce the notepad metaphor. The user manipulates objects displayed on the notepad by drag&drop gestures, which are interpreted uniformly as function application. The system can thus be used for any application that provides concrete representation types for objects and applicable operations. Aspinall and Lüth [3] implement a generic interface that can be configured with the command templates to be filled out for drag&drop gestures.

The $\text{LOUI}$ interface [28] for $\Omega$mega represents rules and tactics introduced by theories by new menu entries. If a tactic requires parameters such as the instantiations for a variable, a generic dialog is opened for the user to enter the missing data.

Broker [11] architectures allow loosely coupled, possibly distributed components to interact by message passing. The $\Omega$mega prover [28] and current versions of the ProofGeneral [5, 4] provide an infrastructure for prover and display components to communicate. The XML-based PGIP protocol [5] abstracts over specific provers and defines a communication standard. The message passing interface in broker architectures requires the individual components to maintain internal state, such that programming extensions, especially with click’n prove functionality, involves a substantial effort.

6 Future Work and Discussion

We are currently developing the implementation of the architecture and a click’n prove interface for Isabelle. The example plugins in Section 4 show the current state and exhibit further immediate requirements: proof script management for multiple documents [9, 5], a View Handler [11] for switching between visualizations, and context-sensitive syntax highlighting. An open strategic question is whether our architecture should build on the Eclipse platform [13, 15] (following [4]) to import the available view management and plugin mechanisms. There are mainly three considerations: a user contributing click’n prove actions should not have to know the Eclipse plugin model in detail, the graphical front-end is
to remain light-weight, and special plugins for single theories may need to be loaded after startup and from locations outside of the installation directory.

A major direction is the integration of new views. Apart from visualizing the goal structure (see for example [19]), we intend to provide the notepad metaphor [21] for offline-calculations, which would be performed by silent prover commands (Section 4.2). This extension requires a new DragAndDropEvent with selected source and selected destination objects as parameters, which is readily provided. We expect that the editing of Isar proof scripts [23] will be simplified if intermediate results in calculational reasoning and auxiliary facts can be generated by forward reasoning on the notepad.

The second major goal is the definition of a stable API for plugins and a more comprehensive set of events. Useful extensions can be found in the motivation of the Interceptor pattern [26]. For instance, mathematical notation can be encoded and decoded by a plugin if the framework provides interpretable events Load and Save for proof scripts, Send for commands, and Receive for answers.

A question that can be answered only with more experience is whether the access to syntax trees is sufficient to implement all desirable click’n prove actions. Bertot [6] provides a partial answer: once a grammar for the term structure is completed, proof-by-pointing can be implemented. However, the Isabelle parsing model [25, §8] defines four steps to obtain the internal tree representation: generation of parse trees, application of parse translations and macro expansion, and finally type inference. We contend that it will be sufficient that parse trees are available, because users necessarily manipulate this externally visible form, thus expecting support only at this level. The lack of type information implies, however, that no disambiguation of parse DAGs can take place and the available actions cannot depend on types.

7 Conclusion

We have presented an architecture for click’n prove interfaces to interactive theorem provers that differs from previous proposals in two aspects: it requires no support from the prover beyond a textual interface with a read-eval-print loop, and it is extensible by light-weight plugins that register to be notified for defined events. Our architecture is based entirely on established patterns for interactive and extensible systems, thus linking the special application of interfaces for theorem provers to more widespread software designs.

We use a parser accepting general context-free grammars to analyze both the proof state and the proof script. Since the parser works incrementally, users can select subterms from the script during editing, but they are not constrained to manipulating only well-formed syntax trees.

The experience of the current implementation suggests that moderately experienced programmers can provide new click’n prove functionality within the given framework. Actions may analyze the structure of the proof state and proof script in detail to decide which commands to generate, they may invoke the prover for more sophisticated queries, and may use arbitrary services of the GUI toolkit for specialized communication with the user.
References


Co-inductive Proofs for Streams in PVS*

Hanne Gottliebsen

Department of Computer Science, Queen Mary, University of London
hago@dcs.qmul.ac.uk

Abstract. We present an implementation in the theorem prover PVS of co-inductive stream calculus. Stream calculus can be used to model signal flow graphs, and thus provides a nice mathematical foundation for reasoning about properties of signal flow graphs, which are again used to model a variety of systems such as digital signal processing. We show how proofs by co-induction are used to prove equality of streams, and present for the first time an strategy to do this automatically.

1 Introduction

How do we reason in an abstract way about signals and circuits? Classical mathematical techniques require analysis relying on numerical methods. An alternative approach is to take a more abstract view, for example using signal flow graphs[1]. This allows for a much more precise model of the circuits, and thus provides a better basis for analysis.

The work we present here is part of the Logical Structures for Control project, which is concerned with reasoning about dynamical systems. The projects has two main strands. One strand is building a Hoare logic for dynamical systems in general[2,3], which can then be applied to various models of dynamical systems, such as block diagrams or signal flow graphs. The other strand is constructing a practical implementation for reasoning about models of dynamical systems. It is within this second strand that the work presented here falls.

It turns out that for several practical research areas, signal flow graphs provide an elegant, modular and scalable notation for modelling. Signal flow graphs were originally introduced by Mason[1] in 1953 for modelling linear networks. Since then, they have been used for a variety of systems, for example modelling circuit transposition for circuits[4], test generation for mixed-signal devices[5] and for digital filters[6]. Thus they are generally used as a modelling tool.

Stream calculus was introduced by Escardó and Pavlović[7]. Their main idea is to interpret the stream elements as factors in approximation for functions, thus establishing the connection between stream calculus and classical calculus. The basic operations of stream calculus are conceptually very simple, for instance differentiation corresponds to taking the tail of a stream, and so stream calculus provides a way to work on problems such as differential equations.

* This work is supported in part by The Nuffield Foundation.
Rutten[8] showed how signal flow graphs can be modelled very nicely using stream calculus. This gives us a precise mathematical notion of what a signal flow graph represents, and allows us to perform mathematical analysis of signal flow graphs in a precise way. Based on this idea of stream calculus as a model for signal flow graphs, we did an implementation in the theorem prover PVS of stream calculus, as we wanted to investigate formal reasoning about signal flow graphs.

PVS (Prototype Verification System)[9] is an interactive theorem prover. The specification language is powerful and allows the use of predicate subtyping as well as higher-order classical logic. Although PVS is an interactive theorem prover, a good level of automation is provided and PVS also facilitates the addition of user-defined automation. Of particular importance to our implementation is the existing support in PVS for co-inductive datatypes. Our implementation of stream calculus in PVS allows us to reason in the most rigorous sense about properties streams and functions over streams.

The implementation is based on co-inductive streams, that is streams are defined as infinite lists rather than as functions from the natural numbers to their element types. We have found that many definitions and also proofs are more intuitive using this technique, although of course conceptually (if not in implementation) the two basic ideas would model the same streams. Using stream calculus allows a very simple way to solve differential equations, and we use this in some of our examples. Many proofs in the implementation relies on the co-induction principle, which in turns requires a bisimulation between two streams to be given. Just like working out an induction hypothesis for a regular proof by induction can be difficult, so can choosing the “right” bisimulation. However, we show how one can systematically “guess” a bisimulation from the proof goal. This is important, as it makes the proofs a lot simpler. We have implemented this as a strategy in PVS and present here for the first time this strategy and some applications of it.

The current implementation is the first step in providing a platform for formal reasoning about signal flow graphs using stream calculus. It is evident that such a platform would be useful for many different topics.

1.1 Structure

In Sect. 2 we explain in some detail the basic concepts of stream calculus. Section 3 outlines the implementation of stream calculus in PVS, with the use of a co-inductive datatype. Section 4 explains how we may automate proofs by co-induction, thus making the current implementation much easier to use. Finally, Sect. 5 contains some conclusions and directions for further work.

2 Stream Calculus

The notion of stream calculus was introduced by Escardó and Pavlović[7] as a means to do symbolic computation (in for example computer algebra and theorem proving) using co-induction.
Streams in general can be defined over any kind of element type, however since we intend to model signal flow graphs, we will restrict our element type to the set $\mathbb{R}$ of real numbers. Following the definitions of Rutten[8], we view a stream as a function from $\mathbb{N}$ to $\mathbb{R}$, and let the set of streams over the reals be denoted by $\mathbb{R}^\omega$:

$$\mathbb{R}^\omega = \{ \sigma | \sigma : \mathbb{N} \rightarrow \mathbb{R} \}$$

Following the tradition of Escardó and Pavlović[7] we use the following terminology: we call $\sigma(0)$ the initial value of the stream $\sigma$, and the derivative $\sigma'$ of the stream $\sigma$ is given by

$$\sigma'(n) = \sigma(n + 1)$$

These are more commonly known as head and tail in computer science, however having this notion of a derivative allows the development of a calculus of streams[7] which is fairly close to that of classical functional analysis. We may use :: to denote appending elements to streams, for example

$$\sigma = a_0 :: a_1 :: \rho$$

where $\sigma$ and $\rho$ are streams and $a_0$ and $a_1$ are stream elements.

We can now define addition and multiplication of streams as follows. The sum, $\sigma + \tau$ of streams $\sigma$ and $\tau$ is element-wise, that is

$$\forall n \in \mathbb{N} : (\sigma + \tau)(n) = \sigma(n) + \tau(n)$$

The convolution product, $\sigma \times \tau$ of streams $\sigma$ and $\tau$ is given by

$$\forall n \in \mathbb{N} : (\sigma \times \tau)(n) = \sum_{k=0}^{n} \sigma(k) \cdot \tau(n - k)$$

We can embed the real numbers into the streams by defining the following stream. Let $r \in \mathbb{R}$. Then $[r]$ is defined as follows:

$$[r] = (r, 0, 0, 0, \ldots)$$

This essentially allows us to add and multiply real numbers and streams:

$$[r] + \sigma = (r + \sigma(0), \sigma(1), \sigma(2), \ldots)$$
$$[r] \times \sigma = (r \cdot \sigma(0), r \cdot \sigma(1), r \cdot \sigma(2), \ldots)$$

Often we will simply use $r$ to denote the stream $[r]$, it will be clear from the context if $r$ is a real number or the stream related to the number.

Finally, we can define a constant stream of particular interest, $X$:

$$X = (0, 1, 0, 0, \ldots)$$

The effect of multiplying a stream by $X$ is a delay of 1, that is:

$$X \times \sigma = (0, \sigma(0), \sigma(1), \sigma(2), \ldots)$$
We see that multiplication by \( X \) is essentially an antiderivative, in the sense that if we multiply a stream by \( X \) and then differentiate, we get the original stream back:

\[
(X \times \sigma)' = \sigma
\]

However, the reverse is only true if the initial value of \( \sigma \) is 0. This corresponds to the constant of integration in analysis being 0.

With the above definitions of differentiation, addition and multiplication, we can obtain the following facts about differentiation of sums and products by applying the basic operations:

\[
(\sigma + \tau)' = \sigma' + \tau'
\]

\[
(\sigma \times \tau)' = ([\sigma(0)] \times \tau') + (\sigma' \times \tau)
\]

We see that the sum behaves exactly as in classical calculus, however multiplication does not.

3 Stream Calculus in PVS

PVS has an extensive set of libraries, and provides easy ways for users to define new datatypes and related operations. In particular, defining a co-inductive datatype is simple with PVS automatically providing the basic operations and properties of the datatype based on the definition. With PVS being strongly typed and supporting predicate subtyping, we can use types to fully define certain functions, something which we use extensively in this implementation.

We have a basic implementation of stream calculus in PVS. The implementation covers all the operations described in the previous section.

3.1 Basic Notion of Streams

We are using the co-inductive datatype constructor in PVS to implement the streams:

```plaintext
stream : CODATATYPE
BEGIN
  str(car: real, cdr:stream):str?
END stream
```

This gives us a datatype `stream` which works essentially like infinite lists, with `car` and `cdr` denoting initial value and derivative, respectively. With the co-inductive datatype we get (for free, from the implementation of `CODATATYPE` in PVS) various theorems, most importantly for us the definition of co-induction:

```plaintext
coinduction: AXIOM
FORALL (B: (bisimulation?), x: stream, y: stream):
  B(x, y) => x = y;
```
In PVS, there is a built-in polymorphic type for sequences, modelled as a function from the natural number to the element type – thus much closer to the definition of streams given in Sect. 2. This of course could also be used to model streams. Indeed, when we first started this work, that is the approach we took[10]. Normally, one would argue in favour of using the co-inductive datatype mainly if one wants to model finite streams alongside the infinite streams. At the moment we are not doing this, although we are considering it. However, we chose to use the co-inductive datatype regardless, since many of the existing proofs (in papers) of properties of stream calculus uses co-induction. In fact, we have basic implementations of stream calculus in both notations, and we found that many proofs are simpler using the co-inductive datatype. We also wanted to investigate closer how the co-inductive datatype works in PVS, and hope that this implementation can serve as an example for PVS users in general. Another reason for choosing to use the co-inductive datatype, is that, with a little work, it allows us to give definitions very close in style to those with self-reference corresponding to:

\[ A = a_0 :: a_1 :: A \]

This is not in general simple in PVS, where items must be declared before they can be used.

Declaring co-inductive streams in PVS, as well as the main result about doing proofs using co-induction, we also get various functions used in constructing co-inductive streams, for example \( \text{inj}_{\text{str}} \) and \( \text{coreduce} \), most of which are somewhat cumbersome to work with in part due to being automatically generated. In order to simplify the notation compared to the automatically generated notation, we introduce our own notation, inspired by[11]:

\[
f, g : \text{VAR} \ [\text{real} \rightarrow \text{real}] \\
a : \text{VAR real} \\
\text{corec}(f,g)(a) : \text{stream} = \\
\text{coreduce}(\lambda (b: \text{real}) : \text{inj}_{\text{str}}(f(b), g(b)))(a)
\]

The meaning of this is that given two functions over the reals \( f \) and \( g \) and a real number \( a \), \( \text{corec}(f,g)(a) \) returns the following stream:

\[ \text{corec}(f,g)(a) = f(a) :: f(g(a)) :: f(g(g(a))) :: \ldots \]

That is, element \( n \) in the stream is defined by successively applying \( g \) and then finally applying \( f \) once. In fact, we defined \( \text{corec} \) and its associated operations and properties in a polymorphic way, as this was useful for implementing certain stream operators.

Now, if we want to be able to define a constant stream such as \( a :: a :: a :: \ldots \), it can be done as follows:

\[
\text{const}(a) : \text{stream} = \text{corec}[\text{real}](\text{id}, \text{id})(a)
\]

where \( a \) is a real number and \( \text{id} \) is the identity function on the real numbers. We can then use co-induction to prove the following lemma, confirming that \( \text{const} \) does indeed give us the constant stream.
const_fact : LEMMA const(a) = str(a,const(a))

There are many functions which can be defined quite nicely using corecursion, for example map and iter (short for iterate):

map(f,s) : stream = corec[stream](lambda t1 : f(car(t1)),
lambda t1 : cdr(t1))(s)

iter(f,a) : stream = corec[real](id, f)(a)

where \( f : \mathbb{R} \rightarrow \text{real} \), \( a : \mathbb{R} \) and \( s \) is a stream. Then \( \text{map}(f,s) \) returns the stream of \( f \) applied element-wise to \( s \), and \( \text{iter}(f,a) \) returns the stream where \( f \) is applied to \( a \) \( n \) times for element \( n \). Thus

\[
\text{map}(f,s) = f(s_0) :: f(s_1) :: f(s_2) :: \ldots
\]
\[
\text{iter}(f,a) = a :: f(a) :: f(f(a)) :: \ldots
\]

3.2 Calculating with Streams

We now define the basic operations on streams:

initial(sigma) : real = car(sigma)

derivative(sigma) : stream = cdr(sigma)

concat(a, sigma) : stream = str(a,sigma)

% Adding two streams
+ (s1,s2) : stream = corec(lambda t1,t2 : car(t1) + car(t2),
lambda t1,t2 : (cdr(t1),cdr(t2)))(s1,s2)

% Scalar multiplication
m(a,s) : stream = map(lambda x : a * x, s)

% Register/Delay
R(s) : stream = str(0,s)

% Some fact to show that the definitions work as expected
add_fact : LEMMA
+(s1,s2) = str(car(s1) + car(s2), +(cdr(s1),cdr(s2)))

m_fact : LEMMA m(a,s) = str(a*car(s), m(a,cdr(s)))

R_inv_deriv : LEMMA derivative(R(s)) = s

One operator which is made slightly more complicated because we are using the co-inductive datatype rather than streams as functions over the natural
numbers, is the convolution product. This really highlights another difference between the two notations. In order to calculate the convolution product, we need *history*, in the sense that for each element of the resulting stream, we need to know all preceding elements of each of the two input streams. There is no way to avoid this. In our implementation we get around this by essentially going back to considering element number \( n \), something which is a bit unnatural in the co-inductive definition, but nonetheless works.

### 3.3 Differential Equations

One of the interesting questions about a given signal flow graph is whether it implements a solution to a certain differential equation or not. Thus, we might consider the following first-order stream differential equation, and ask what \( s \) should be:

\[
\begin{align*}
s(0) &= a \land s' = s
\end{align*}
\]

(1)

Just by looking at the stream and remembering the definition of the derivative of a stream, we get

\[
\begin{align*}
s &= a :: s \\
  &= a :: a :: s \\
  &= a :: a :: a :: s
\end{align*}
\]

So we see that \( s \) is the constant stream over \( a \). Then if our signal flow graph outputs this stream, it gives a solution to the differential equation.

In general, a first-order stream differential equation is of this form:

\[
\begin{align*}
\tau(0) &= a \land \tau' = \sigma
\end{align*}
\]

(2)

where \( a \) is some real number and \( \tau \) and \( \sigma \) are streams of real numbers. This leads us to the following definition in PVS of the solution to a first-order differential equation:

\[
\text{fode}(s,a) : \text{stream} = \{ t \mid \text{car}(t) = a \land \text{cdr}(t) = s \}
\]

This means that that the function \( \text{fode} \) takes arguments \( s \) and \( a \) and returns a single stream with initial value \( a \) and derivative \( s \). That is, \( \text{fode}(s,a) \) is the solution to (2). So we immediately have the following result:

\[
\text{fode}_\text{fact} : \text{LEMM}A \\
\text{fode}(s,a) = \text{str}(a,s)
\]

We can now prove the result of our example (1) above:

\[
\text{example}_\text{i.3} : \text{LEMM}A \\
s = \text{fode}(s,a) \implies s = \text{const}(a)
\]

We have implemented second-order differential equations in essentially the same manner:
sode(s,a,b) : stream = { t | car(t) = a and car(cdr(t)) = b and cdr(cdr(t)) = s }

sode_fact : LEMMA
  sode(s,a,b) = str(a,str(b,s))

As an example, consider the following second-order stream differential equation:

\[
\begin{align*}
s(0) &= a \\
s(1) &= b \\
s'' &= s
\end{align*}
\]

Again, by looking at the stream, we can find s:

\[
\begin{align*}
s &= a :: b :: s \\
   &= a :: b :: a :: b :: s
\end{align*}
\]

We see that this corresponds to the constant streams over a and b respectively zipped, in PVS:

example_1_4 : LEMMA
  s1 = sode(s1,a,b) IMPLIES s1 = zip(const(a),const(b))

where zip(s,t) = s_0 :: t_0 :: s_1 :: t_1 :: ....

To summarise, we can define streams and functions over streams. We can also do basic interactive proofs in PVS, but in the next section we will discuss new automation for proofs by co-induction, making most of the proofs of general lemmas about streams and functions over streams much easier.

4 Automation of Co-inductive Proofs

In this section we will discuss co-induction in general, and see how we can automate proofs by co-induction in PVS. The strategy presented here is set up specifically to work with the implementation of stream calculus, but we see no reason why it cannot be generalised to co-inductive datatypes in general.

Following[8] we first introduce formally the notion of bisimulation on streams and the co-induction principle for streams. A bisimulation is a relation, defined as follows:

**Definition 1 (Bisimulation).** Let R be a binary relation on streams. Then R is a bisimulation if for all streams s and t the following holds:

\[ s R t \Rightarrow \text{car}(s) = \text{car}(t) \land \text{cdr}(s) R \text{cdr}(t) \]

We see that equality is an obvious bisimulation. However, not all bisimulations are equivalent to equality, for example the relation

\[ R(s,t) = (s = \text{const}(1) \land t = \text{const}(1)) \]
is a bisimulation, but the stream \((1,0,0,\ldots)\) is not related to itself under this bisimulation, although it is obviously equal to itself. In PVS the above definition is generated automatically when declaring a co-inductive datatype, such as our streams.

Let us remind ourselves about the co-induction principle as seen in Sect. 3.1:

**Theorem 1 (Co-induction).**

Let two streams, \(s\) and \(t\), and a bisimulation on streams, \(\equiv\), be given. Then, if \(s \equiv t\) then \(s = t\).

Again, this theorem is generated automatically by PVS for any co-inductive datatype, obviously with the appropriate types used.

Every time we want to do a proof by co-induction, we need to “invent” a bisimulation which fits the statement we are trying to prove. This is not necessarily a simple task, and as usual with theorem proving, having better support for automation makes proving so much easier. Let us look at an example:

**Example 1 (Proof by Co-induction – map_iter).**

We want to prove that for the functions \(\text{map}\) and \(\text{iter}\) (Sect. 3.1), a function \(f : \mathbb{R} \to \mathbb{R}\) and a real number \(x\), we have the following:

\[
\text{map}(f, \text{iter}(f, x)) = \text{iter}(f, f(x))
\]

Since both \(\text{map}\) and \(\text{iter}\) return streams, this is an equality between streams.

Now consider the relation \(R\), where

\[
R(s, t) = \exists g : \mathbb{R} \to \mathbb{R}, y : \text{real} : s = \text{map}(g, \text{iter}(g, y)) \land s = \text{iter}(g, g(y))
\]  

(3)

It is clear that \(R(\text{map}(f, \text{iter}(f, x)), \text{iter}(f, f(x)))\) since we can choose \(f\) for \(g\) and \(x\) for \(y\). So, assuming that \(R\) is indeed a bisimulation, we can use Theorem 1 to prove equality of the two stream.

Thus we need to prove that \(R\) is indeed a bisimulation, that is \(sRt \Rightarrow \text{car}(s) = \text{car}(t) \land \text{cdr}(s) = \text{cdr}(t)\) for all streams \(s\) and \(t\). This can be easily done by unfolding the definitions of \(\text{map}\) and \(\text{iter}\).

In the above example, we see that given an appropriate relation, in this case \(R\) as in (3), the rest of the proof is relatively simple, if tedious. So it seems that the step, when doing proof by co-induction, which requires “invention” is determining which relation to use. However, it is possible to do this in a very mechanistic way for many cases stream equality. Looking again at the above example, we see that the relation suggested can be obtained simply by considering the structure of the equality, we want to prove. One may wonder why we need to introduce new, existentially quantified variables \(g\) and \(y\) in the definition of the relation. The reason for this is to ensure that the relation is indeed a bisimulation, for which we need the cdr’s of two related streams to also be related.

In Ex. 1 the proof of the relation being a bisimulation is quite simple, using only rewriting and simplification, but in some cases this step requires real insight and relies on using other lemmas about properties of the functions and streams involved. For this reason, this step has not yet been automated.
The simple pattern matching of the actual formula being proven illustrated above is not in general enough. Often, in a proof, we have various assumptions which may also be used in the proof, and the very basic relation above does not capture those. This means that when it comes to proving that the relation is a bisimulation, this may not be possible. One such example is (1). So, as well as looking at the actually equality we want to prove, we also need to consider the rest of the current proof goal.

4.1 The PVS strategy

The basic structure of the PVS strategy \textit{bisim} is as follows:

\textbf{Collect Variables} In order to determine which variables, and their types, to have existentially quantified in the relation, we construct a list of the original variables, their types and new variables, which will be of the same type.

\textbf{Assumptions} We need to decide which assumption needs to be added to the relation. This is done based on the occurrence of any of the variables found above. That is, if an assumption contains any of the variables collected, then a version of this assumption, with suitable substitutions, should be included in the relation.

\textbf{Build Relation} We use the original formulae, but substitute with our new variables to build the string used to instantiate the Theorem of Co-induction.

\textbf{Instantiate Theorem of Co-induction} We then instantiate the theorem, using the relation just built and the original streams also.

\textbf{Prove the Assumption} Since the theorem says $R(s, t) \Rightarrow s = t$, we need to prove that the relation holds on the two original streams. However, since the relation is built based on pattern matching with the original streams, this is easily done using the “obvious” instantiation.

\textbf{Prove that Relation is a Bisimulation} Finally, we need to prove that the relation is a bisimulation. This is done by proving each of the statements on the car’s being equal and the cdr’s being related.

We see that most of the work of this strategy goes into setting up terms to be used in the instantiation of the theorem. In fact, the actual PVS proof does only the following: skolemize, introduce the theorem, instantiation (both the theorem, as explained above, and automatic instantiation provided by PVS), replacing terms and simplifying. Other than the instantiation of the theorem, none of these requires any particular insight, and indeed this part of the automation is very simple.

\textit{Example 2 (PVS proof of map\textunderscore iter).}

Consider again the statement

\[
\text{map}(f, \text{iter}(f, x)) = \text{iter}(f, f(x))
\]

where $f : \mathbb{R} \rightarrow \mathbb{R}$ and $x : \mathbb{R}$.

In PVS, we can prove this by using our strategy \textit{bisim}:
map_iter :
{1}  FORALL (f: [real -> real], x: real):
    map(f, iter(f, x)) = iter(f, f(x))

Remember that bisim leaves us to prove only the two criteria for the relation being a bisimulation. So the two subgoals produced here correspond to the two parts of the definition of a bisimulation.

The first subgoal is then

map_iter.1 :
{1}  car(map(g, iter(g, y))) = car(iter(g, g(y)))

This is then completed by expanding map and iter to their definitions in terms of corec and then using rewrites about car and corec.

The second subgoal is

map_iter.2 :
{1}  EXISTS (g1: [real -> real], y1: real):
    cdr(map(g, iter(g, y))) = map(g1, iter(g1, y1))
    AND cdr(iter(g, g(y))) = iter(g1, g1(y1))

Again, we expand map and iter and use rewrites about cdr and corec. After this PVS is able to automatically guess the correct instantiation of g1 = g and y1 = g(y) to complete the proof. If we attempt to instantiate before the expansion, PVS will guess the wrong instantiation in this case, thus the expansion should take place first.

This completes the proof.

In both subgoals, new variables generated by the strategy is used. For the sake of readability we have substituted simpler names above.

4.2 Applicability of Strategy

It is clear that the strategy is quite basic, however it works quite well on typical examples of equalities which occur in for example [8]. There are some areas where the strategy does not perform so well:

Foundational Statements For some foundational statements, which are part of the basic definitions and properties for streams, it turns out that we cannot prove that the generated relation is a bisimulation. In these cases we often need a relation which is stronger than that provided by the strategy.

Streams as Sequences For streams or functions viewed as sequences (functions over the natural numbers), it is often more appropriate to use element-wise equality as the bisimulation, however this is not produced by the strategy.
Complicated Bisimulations In some cases, the relation is such that the proof that it is a bisimulation relies on other lemmas. In many cases, it is possible by systematic rewriting and simplification to present the goal to PVS in a form where PVS can make the correct instantiation, however this is not always possible. If instantiation by the user is needed, we then have a problem with the automatically generated variable names, since for a rerun of the proof, these will be different. Thus a proof constructed this way will not work if rerun.

Others There are some statements which are just not on the right form for this strategy to work. One example is the following:

\[
\text{double}_\text{zip} : \text{LEMMA double}(s) = \text{zip}(s, s)
\]

where \(\text{double}(s)\) duplicates each element in \(s\), that is \(\text{double}(s) = s_0 :: s_0 :: s_1 :: s_1 :: \ldots\). In this case, the relation needs to look two elements ahead rather than only one, something not accounted for in the strategy.

There are various ways we can try to alleviate the above problems. For example, the problem of the naming and renaming of fresh variables might be handled differently, allowing us to fix the names locally within each proof, and so prevent this particular problem of not being able to rerun proofs automatically. In fact, with the current version it is possible to get around this by using the glass box version of the bisim strategy. This stores as the proof the individual proof steps and thus keeps a record of the instantiation involving the variable names, keeping them the same for any subsequent reruns of the proof. For statements where a straightforward proof by co-induction is not feasible, we can set up other, related proof strategies. It turns out that the lemma \(\text{double}_\text{zip}\) is can be proven using what we might call even-odd co-induction: Essentially, if we have a bisimulation between the even elements of two streams and a bisimulation between the odd elements, then the two streams are equal. We have proven this principle in PVS and used it to prove \(\text{double}_\text{zip}\) in a manner very similar to that of using bisim. However, bisim does not at this time handle such proofs. Based on our study of the examples used in [8] it seems that it may be beneficial to support automatic proof using element-wise equality as the bisimulation. In many cases using this would overly complicate the proof, essentially converting to a sequence notation for the streams, however in some cases it works well, particularly where the stream (or any functions) involved are defined by considering streams as sequences.

We will be using the first-order differential equation from (1) as an example, and want to prove that \(s = \text{const}(a)\), when \(s = \text{fode}(s, a)\). In any given proof, we may have not only the actual equality that we are trying to prove, but also other equalities for the streams, which we may use as assumption in our proof. If we have assumptions of the form \(s = s_1(x_0, \ldots, x_n)\) (and similarly for \(t\)), we take as the general bisimulation the following function:

\[
\lambda s_2, t_2 : \exists x, y, s_1, t_1 : s_2 = s \land s = s_1(x) \land t_2 = t \land t = t_1(y)
\]
For our example, this becomes
\[ \lambda s2, t2 : \exists a, s1 : s2 = s1 \land s1 = fode(s1, a) \land t2 = const(a) \]
as there are no extra assumption on \(const(a)\). In this particular case, it looks like the term \(s2 = s1\) is not necessary; however this method is for the general case where \(s1\) is a compound term, in which case this extra term is needed.

After having found the “correct” relation to use, the co-induction theorem gives us the equality between the streams. However, we still need to prove that the relation is indeed a bisimulation. Of course, any bisimulation would work, but for the sake of completing the proof as painlessly as possible, it makes sense to choose one which is not too difficult to handle. The method outlined above for finding a bisimulation does not necessarily lead to the smartest, simplest or indeed the most intuitive bisimulation, but so far it has proven to work quite well.

Let us consider a more complex example using PVS. In this case, we are looking at the second-order differential stream equation from Sect. 3.3. After some initial steps, we have the following proof goal to work on:

\[ \text{FORALL } (B : (\text{bisimulation}?), x : \text{stream}, y : \text{stream}): \]
\[ B(x, y) \Rightarrow x = y \]
\[ s1!1 = \text{sode}(s1!1, a!1, b!1) \]
\[ \text{zip}(\text{const}(a!1), \text{const}(b!1)) \]

We need to prove that formulae -1 and -2 implies formula 1.

We can now instantiate formula -1 according to the rules set out above:

\(\text{inst } -1 \text{ "lambda } (s2, t2) : \exists (s1, a, b) : s2 = s1 \land s1 = \text{sode}(s1, a, b) \land t2 = \text{zip}(\text{const}(a), \text{const}(b))"\)
\(\text{s1!1} \text{ "zip}(\text{const}(a!1), \text{const}(b!1))")\)

This gives us the following goal:

\[ \text{EXISTS } (s1, a, b): \]
\[ s1!1 = s1 \land s1 = \text{sode}(s1, a, b) \land \text{zip}(\text{const}(a!1), \text{const}(b!1)) = \text{zip}(\text{const}(a), \text{const}(b)) \]
\[ \Rightarrow s1!1 = \text{zip}(\text{const}(a!1), \text{const}(b!1)) \]
\[ s1!1 = \text{sode}(s1!1, a!1, b!1) \]
\[ \text{zip}(\text{const}(a!1), \text{const}(b!1)) \]

where formula -1 has as a consequent exactly the thing we are trying to prove in formula 1, and the assumption in formula -1 is trivially true if we choose the obvious instantiations for \(s1, a\) and \(b\), something which PVS will do for us automatically.

The other part of the proof is then that the relation we entered before must be a bisimulation:
\[-1\] \( s1!1 = sode(s1!1, a!1, b!1) \)
\[\]-------
\{1\} \quad \text{bisimulation?}(\text{LAMBDA} (s2, t2):
  \quad \text{EXISTS} (s1, a, b):
  \quad s2 = s1 \text{ AND}
  \quad s1 = sode(s1, a, b) \text{ AND}
  \quad t2 = \text{zip}(\text{const}(a), \text{const}(b)))
\[2\] \quad s1!1 = \text{zip}(\text{const}(a!1), \text{const}(b!1))

The proof of this is a bit longer and more tedious, however no great insight is required, so it is now quite simple. Thus using the method of guessing a bisimulation has helped us complete the proof.

5 Conclusions

One of the main discussion points in this project has been which underlying datatype to use for the streams: functions over natural numbers or a co-inductive datatype. Of course, using only infinite streams, the two datatypes are equivalent, but due to the datatypes and the support for definitions and proofs, the choice does matter. Having tried both, we have concluded that using the co-inductive datatype leads to more natural definitions (once past the initial less pretty automatically generated constructors) and proofs.

We have shown that with our implementation, we can define and solve stream differential equations amongst other things. In general, stream calculus is simply, but elegant and thus provides a very neat application for mechanised reasoning.

One of the main issues of the use of an implementation in a theorem prover is automation, since this has a big impact on efficiency and accessibility. We have addressed this through our implementation of the strategy \textit{bisim}, which for many equalities guess the correct bisimulation to use for a proof by co-induction. It is clear that our strategy works well for a large class of equations over streams, however we have also identified some classes where it fails and given suggestions for possible solutions in these cases.

The next stage of our project is to do a case study of a signal flow graph, for instance one modelling a filter. This would show the strength of the implementation as a tool for formal analysis of a very practical application of signal flow graphs. PVS already has extensive libraries for classical functional analysis, and we intend to also connect the notion of streams as representations of Taylor series expansions to the existing analysis libraries in PVS.

6 Acknowledgements

We would like to thank César Muñoz for his assistance with the initial version of the strategy \textit{bisim}, Sam Owre for help in understanding how the co-inductive datatypes in PVS work. Thanks also to Paul Miner for his insight on using co-induction in general and on the usefulness of the function \textit{corec} in particular.
References

A Code Generator Framework for Isabelle/HOL

Florian Haftmann* and Tobias Nipkow
Institut für Informatik, Technische Universität München
http://www.in.tum.de/\{∼haftmann,∼nipkow\}

Abstract. We present a code generator framework for Isabelle/HOL. It formalizes the intermediate stages between the purely logical description in terms of equational theorems and a programming language. Correctness of the translation is established by giving the intermediate languages (a subset of Haskell) an equational semantics and relating it back to the logical level. To allow code generation for SML, we present and prove correct a (dictionary-based) translation eliminating type classes. The design of our framework covers different functional target languages.

1 Introduction and related work

Executing formal specifications is a well-established topic and many theorem provers support this activity by generating code in a standard programming language from a logical description, typically by translating an internal functional language to an external one:

- Coq [15] can generate OCaml both from constructive proofs and explicitly defined recursive functions.
- Both Isabelle/HOL [1] and HOL4 generate SML code. In the case of Isabelle this code is also used for counter example search [2].
- The language of the theorem prover ACL2 is a subset of Common Lisp.
- PVS allows evaluation of ground terms by translation to Common Lisp [4].

Though code generation forms an increasingly vital part of many theorem provers, its functionality is often not formalized and must be trusted. In the case of ACL2 this is justified because its logic is a subset of Common Lisp, but the addition of single-threaded objects [3], which allow destructive updates, breaks this direct correspondence. The treatment of destructive updates in the Common Lisp code generated by PVS has been proved correct [14], although PVS code generation in general appears to have not been formalized. Code generation for Coq is studied in great detail, e.g. [7]. One of the key differences to our work is that Coq is already closer to a programming language than HOL, for example because it has inductive types built in. Code generation for Isabelle/HOL is described by Berghofer and Nipkow [1], who consider in particular the generation of Prolog-like code from inductive definitions, which we ignore, but who ignore

* Supported by DFG project NI 491/10-1
the correctness question, dismiss the purely functional part as straightforward, and do not cover type classes at all.

The key contributions of our paper can be summarized as follows:

– A framework that formalizes the intermediate stages between the purely logical description in terms of equational theorems and a programming language. Giving the intermediate languages (fragments of Haskell and SML) an equational semantics has two advantages:
  • Correctness of the translation is established in a purely proof theoretic way by relating equational theories.
  • Instead of a fixed programming language we cover all functional languages where reduction of pure terms (no side effects, no exceptions, etc) can be viewed as equational deduction: Given a pure program \( P \) and a pure term \( t \), we assume that if \( t \) reduces to a value \( u \), then \( t \) and \( u \) are equivalent modulo the equations of \( P \). This requirement is met by languages like SML, OCaml and Haskell, and we only generate pure programs.

– A first treatment of code generation for type classes. Although we follow Haskell’s dictionary passing style, the key difference is that in Haskell, type classes are defined by translation, whereas our starting point is a language with type classes which already has a meaning. Thus we need to show that the translation is correct.

– A constructive proof how to transform a set of equations conforming to some implementability restrictions into a program.

Note that throughout this paper type classes refer to their classical formulation (see, for example, [6]). Note further that we can lump Haskell and SML together because we will only guarantee partial correctness of the generated code.

Somewhat related but quite different is the work by Meyer and Wolff [9]. They translate between shallow embeddings of functional programs in HOL by means of tactics, whereas we justify the translation once and for all, but do this outside HOL. There are many further differences, like our thorough treatment of type classes.

After a sketch of the system architecture (§2), we introduce Isabelle’s language of terms and types (§3), accompanied by an abstract Haskell-like programming language and its equational semantics (§4). Type classes are eliminated in favor of dictionaries and this translation into an SML-like sublanguage is proved correct (§5). We characterize implementable equation systems and give a correct translation (via a constructive proof) into the Haskell-like language (§6). A note on handling equality (§7) concludes our presentation.

2 System architecture

Conceptually, the process of code generation is split up in distinct steps:
1. Out of the vast collection of theorems proven in a formal theory, a reasonable subset modeling an equation system is selected.
2. The selected theorems are subjected to a deductive preprocessing step resulting in a structured collection of defining equations.
3. These are translated into a Haskell-like intermediate language.
4. From the intermediate language the final code in the target language is serialized.

Only the two last steps are carried out outside the logic; by making this layer as thin as possible, the amount of code to trust is kept minimal.

3 The Isabelle framework

The logic Isabelle/HOL [10] is an extension of Isabelle’s meta logic, a minimal higher-order logic of simply typed lambda terms [12]. Propositions are terms of a distinguished type \( \text{prop} \). Theorems are propositions constructed via some basic inference rules. Isabelle provides equality \( \equiv \) of terms up to \( \alpha\beta\eta \) conversion.

Isabelle’s term language is an order-sorted typed \( \lambda \)-calculus with schematic polymorphism\(^1\). It has three levels with the following syntax:

- **sorts** \( s ::= c_1 \cap \ldots \cap c_n \)
- **types** \( \tau ::= \kappa \tau_m | \tau_1 \rightarrow \tau_2 | \alpha :: s \)
- **terms** \( t ::= f [\tau_n] | x :: \tau | \lambda x :: \tau. t | t_1 t_2 \)

The notation \( \pi_n \) denotes the tuple or sequence \( u_1, \ldots, u_n \); writing \( \pi \) means the length is irrelevant.

The atomic symbols are classes \( c \), type constructors \( \kappa \), type variables \( \alpha \), constants \( f \), and variables \( x \). Classes classify types. Sorts are intersections of finitely many classes. Type variables are qualified by sorts, which restrict possible type instantiations. The System F-like notation \( f [\tau_n] \) is explained below.

Note that our terms and types have type and sort information attached to each occurrence of a variable. This simplifies the context in the well-formedness judgments to come but means that we tacitly assume that these attachments are consistent within a term or type. The empty intersection represents the universal sort which every type belongs to. Constants have generic types of the form \( \forall \alpha \in \mathcal{A} \cdot \tau \), where \( \{\alpha_1 \ldots \alpha_k\} \) is the set of all type variables in \( \tau \).\(^2\) If all \( s_i \) are empty, we write \( \forall \alpha \in \mathcal{A} \cdot \tau \) instead.

---

\(^1\) Hindley-Milner let-polymorphism without a local let

\(^2\) Whenever a type scheme is given, this restriction is implicit.
A context $\Gamma$ is a four-tuple $(TYP, SUP, \Sigma, \Omega)$ of (partial) functions: $TYP \kappa = k$ means that type constructor $\kappa$ has arity $k$, $SUP c = \overline{c}_q$ that $\{c_1, \ldots, c_q\}$ is exactly the set of direct superclasses of class $c$, and $\Omega f = \forall \alpha :: \overline{s}_k. \tau$ that constant $f$ has generic type $\forall \alpha :: \overline{s}_k. \tau$. The behavior of type constructors w.r.t. classes is expressed by $\Sigma: \Sigma(\kappa, c) = \overline{s}_k$ is called an instance and means that $\kappa \overline{s}_k$ is of class $c$ if each $\tau_i$ is of sort $s_i$. Contexts must not contain $SUP$-cycles. Notation: Every $\Gamma$ is implicitly of the form $(TYP, SUP, \Sigma, \Omega)$; we have the following judgments:

- Well-formed sorts: $\Gamma \vdash s$ means that $s = c_1 \cap \ldots \cap c_n$ is well-formed, i.e. $SUP c_i$ is defined for all $i$.
- Subsort relation: $\Gamma \vdash s \sqsubseteq s'$ means that $s = c_1 \cap \ldots \cap c_n$ is a subsort of $s' = c'_1 \cap \ldots \cap c'_m$, i.e. for all $i$ there is a $j$ such that $c'_j$ is a (not necessarily direct) superclass (w.r.t. $SUP$) of $c_j$.
- Well-formed types: $\Gamma \vdash \tau$ means that all type constructors $\kappa$ in $\tau$ are applied to the required number of arguments $TYP \kappa$.
- Well-sorted types $\Gamma \vdash \tau :: s$ and well-typed terms $\Gamma \vdash t :: \tau$.

Precise definitions can be found elsewhere [11]. An alternative way of defining $\Gamma \vdash t :: \tau$ can be found in §5. The definition of $\Gamma \vdash t :: \tau$ is standard except for constants: If $\Omega f = \forall \alpha :: \overline{s}_k. \tau$ and $\Gamma \vdash \tau_i :: s_i$ for all $i$ then $\Gamma \vdash f[\overline{\tau}_k] :: \tau[\overline{s}_k/\overline{s}_k]$. Each occurrence of a constant in a term carries the instantiation of its generic type to resolve ambiguities due to polymorphism and overloading.

To guarantee principal types, Isabelle enforces coregularity of contexts:

**Definition 1** $\Gamma$ is coregular if $\Sigma(\kappa, c) = \overline{s}_k$ implies $\forall d \in SUP c. \overline{s}_k = \overline{s}_k \land \forall 1 \leq i \leq k. \Gamma \vdash s_i \sqsubseteq s'_i$

It will turn out that coregularity is also essential for dictionary construction (§5).

4 From logic to programs

On the surface level, Isabelle/HOL offers all the constructs one finds in functional programming: data types, recursive functions, classes, and instances. But in the logical kernel, all we have are contexts and theorems. Clearly, equational theorems can represent programs. For example, a suitable context with natural numbers, pairs, class $c$ and a constant $f :: \forall \alpha :: c. \alpha \to \alpha$, and two equations

$$
\begin{align*}
& f[nat](n :: nat) \equiv n + 1 \\
& f[\alpha :: c \times \beta :: c](x :: \alpha, y :: \beta) \equiv (f[x] x, f[y] y)
\end{align*}
$$


have a straightforward interpretation as a Haskell program. But not every set of equations corresponds to a program, even if they superficially look like one:

$$
\begin{align*}
& f[bool \times bool](x :: bool, y :: bool) \equiv (\neg x, \neg y) \\
& f[nat \times nat](x :: nat, y :: nat) \equiv (y, x)
\end{align*}
$$

is also definable in Isabelle, but realization of this overloaded system would demand a sophisticated type class system beyond Haskell 1.0. There are many further restrictions on what is executable. A major part of our task is to establish a precise link between a subset of our logic and an executable language.
Definition 2 An equation of the form \( f[\tau_k]t_m \equiv t \) is a defining equation for \( f \) with type arguments \( [\tau_k] \), arguments \( t_m \) and right hand side \( t \) iff

1. all variables of \( t \) occur in \( t_m \),
2. all type variables of \( t \) occur in \( [\tau_k] \),
3. no variable occurs more than once in \( t_m \) (left-linearity),
4. all \( t_m \) are pattern terms, where a pattern term is either a variable or a constant applied to a list of pattern terms.

Due to these syntactic restrictions, defining equations resemble the kind of equations admissible in function definitions in functional programming languages; we will use defining equations as an abstract “executable” view on the logic:

Definition 3 An equation system is a pair \((\Gamma,E)\) where \( E \) is a set of defining equations which are well-typed in the context \( \Gamma \). We write \((\Gamma,E) \vdash s \equiv t \) iff \( s \equiv t \) follows by equational logic and \( \beta\eta \)-conversion.

Note that \( \Gamma \) is necessary to restrict derivations to well-typed terms.

An abstract programming language

Our aim is to implement a given equation system as a program in a functional programming language. Therefore we introduce a Haskell-like intermediate language which captures the essence of target languages and give it a semantics as an equation system (equational reasoning is a common device in the Haskell community [5]). The language forms a bridge between logical and operational world. Its four statements are \texttt{fun}, \texttt{data}, \texttt{class} and \texttt{inst}. The semantics of statements is given by rules \((\Gamma,\texttt{stmt}) \longrightarrow (\Gamma',E)\) where \( \Gamma' \) denotes an initial context, \( \Gamma' \) the resulting context (an extension of \( \Gamma \)) and \( E \) the set of defining equations.

The reading of the semantics can also be reversed. Then it describes what are the required equational theorems needed as witnesses to generate some statement (see §6). An example program is shown in the left column of Table 2.

Before we go into details we need to discuss the correctness issue. In the end, we want to ensure that if we translate an Isabelle/HOL theory via our intermediate language into SML or Haskell, and the compiler accepts it and some input term \( t \), and reduces \( t \) to \( v \), then \( t \equiv v \) should be provable in Isabelle/HOL. This is partial correctness. Hence we only need to ensure that any equation used during reduction is contained in the semantics \( E \) we give to each statement. This is not hard to check by inspecting our semantics.

The semantics is expressed as a relation because it is partial due to context conditions. These context conditions are more liberal than in SML; this may lead to untypeable SML programs, for example due to polymorphic recursion, but is not a soundness problem.

Extending the various components of \( \Gamma \) is written \( \Gamma[x := y] \) where the component in question is determined by the type of \( x \) and \( y \). For example, \( \Gamma[\kappa := k] \) extends the \texttt{TYP}-component. The extension \( \Gamma[x := y] \) is not permitted if \( x \) is already defined in the corresponding component of \( \Gamma \).
**Function definitions** Function definitions have the syntax

\[
\text{fun } f :: \forall \alpha :: s_k. \tau \text{ where } f t_1 = t_1 | \ldots | f t_n = t_n
\]

where each equation \( f t_i = t_i \) must conform to the restrictions of Definition 2.

The semantics is the obvious one:

\[
\Gamma' = \Gamma[f := \forall \alpha :: s_k. \tau] \quad \forall 1 \leq i \leq n. \quad \Gamma' \vdash f [\alpha :: s_k] t_i \equiv t_i :: \text{prop}
\]

\[
\langle \Gamma, \text{fun} \ldots \rangle \rightarrow \langle \Gamma', \{f[\alpha :: s_k] t_1 \equiv t_1, \ldots\} \rangle
\]

Note that we allow polymorphic recursion.

**Data types** Recursive data types introduce a type constructor \( \kappa \) and term constructors \( f_i \):

\[
data \kappa \overline{\tau}_k = f_1 \text{ of } \overline{\tau}_1 | \ldots | f_n \text{ of } \overline{\tau}_n
\]

The \( \overline{\tau}_k \) must be distinct and no other type variable may occur in the \( \overline{\tau}_i \):

\[
\Gamma' = \Gamma[\kappa := k][f_1 := \forall \alpha :: \overline{\tau}_k \rightarrow \kappa \overline{\tau}_k, \ldots] \quad \forall 1 \leq i \leq n. \quad \Gamma' \vdash \overline{\tau}_i \rightarrow \kappa \overline{\tau}_k
\]

\[
\langle \Gamma, \text{data} \ldots \rangle \rightarrow \langle \Gamma', \{\} \rangle
\]

Note that our actual implementation allows fun to define mutually recursive functions and data mutually recursive data types. The corresponding modifications of our presentation are straightforward but tedious.

**Type classes** Overloading is covered by Haskell-style class and instance declarations [6]:

\[
\text{class } c \subseteq c_1 \cap \ldots \cap c_m \text{ where } f_1 :: \forall \alpha. \tau_1, \ldots, f_n :: \forall \alpha. \tau_n
\]

\[
\text{inst } \kappa \overline{\tau}_k :: c \text{ where } f_1 = t_1, \ldots, f_n = t_n
\]

Class declarations merely extend the context, provided they are well-formed:

\[
\Gamma \vdash c_1 \cap \ldots \cap c_m \quad \forall 1 \leq i \leq n. \quad \Gamma \vdash \tau_i
\]

\[
\langle \Gamma, \text{class} \ldots \rangle \rightarrow \langle \Gamma[c := \overline{\tau}_m][f_1 := \forall \alpha :: c. \tau_1, \ldots], \{\} \rangle
\]

\[
\Gamma' = \Gamma[(\kappa, c) := \overline{\tau}_k]. \quad \forall 1 \leq i \leq n. \quad \exists \alpha. \Omega f_i = \forall \alpha :: c. \tau \wedge \Gamma' \vdash t_i :: \tau[\alpha :: \overline{\tau}_k/\alpha]
\]

\[
\langle \Gamma, \text{inst} \ldots \rangle \rightarrow \langle \Gamma', \{f_1[\alpha :: \overline{\tau}_k] t_1 \equiv t_1, \ldots\} \rangle
\]

where \( \Gamma' \) must be coregular. Note that functions in a single inst may be mutually recursive.

**Programs** Now we lift \( \rightarrow \) to lists of statements:

\[
\langle \Gamma, \text{stmt} \rangle \rightarrow \langle \Gamma', E \rangle \quad \langle \Gamma', \text{stmts} \rangle \rightarrow \langle \Gamma'', E' \rangle \quad \langle \Gamma, \text{stmt; stmts} \rangle \rightarrow \langle \Gamma'', E \cup E' \rangle
\]

\[
\langle \Gamma, [] \rangle \Rightarrow \langle \Gamma, \{\} \rangle \quad \langle \Gamma, (\text{stmt; stmts}) \rangle \Rightarrow \langle \Gamma'', E \cup E' \rangle
\]

Statements are processed incrementally in an SML-like manner.
**Definition 4** A program $S$ is a list of statements with a well-defined semantics, i.e. there exists a transition $(\Gamma_0, S) \Rightarrow (\Gamma, E)$ where $\Gamma_0$ is the initial context containing only the type $\text{prop}$ with polymorphic equality $\equiv$ of result type $\text{prop}$. We then write $S \sim (\Gamma, E)$.

Note that if $S \sim (\Gamma, E)$, each equation in $E$ is well-typed with respect to $\Gamma$ due to monotonicity of context extensions and the construction of the rules for $\Rightarrow$.

Context conditions imposed on $\text{inst}$ statements ensure that $\Gamma$ is coregular.

In §6 we show how to distill a program from a set of equations still involving classes. This requires some type class technicalities presented in §5 where we show how to replace classes by dictionaries.

**5 Dictionary construction**

We have given $\text{inst}$ statements a semantics in terms of overloaded defining equations. In classical Haskell their semantics is given by a dictionary construction [16]. To justify this link, we formalize dictionary construction as a transformation of a program $S$ within the intermediate language to a program in the same language but without any $\text{class}$ or $\text{inst}$ statements. To generate code for target languages lacking type classes (e.g. SML), this construction is carried out on the intermediate language (i.e. outside the logic).

Within our framework of order-sorted algebra, dictionary construction is described as a translation (relative to some $\Gamma$) of order-sorted types into dictionary terms [17], lifted to terms and statements:

- $(\tau :: c)$ maps a well-sortedness judgment to a corresponding dictionary,
- $(t)$ introduces dictionaries into a term $t$,
- $(S)$ transforms a program to its typeclass-free counterpart.

Dictionaries are built from (global) dictionary constants $c_\kappa$ and (local) dictionary variables $\alpha_n$ with explicit projections $\pi_{d \to c}$ by the following interpretation of rules for well-sortedness judgments. Dictionary construction relies on a unique representation of sorts. When writing $c_1 \cap \ldots \cap c_m$ we assume a total order of classes and that $c_1 \cap \ldots \cap c_m$ is minimal, i.e. no $c_i$ is a subclass of any other $c_j$.

\[
\begin{align*}
(\tau :: c_1) &= D_1 \quad \ldots \quad (\tau :: c_q) = D_q & \text{(sort$_D$)} \\
(\tau :: c_1 \cap \ldots \cap c_q) &= D_q \\
(\tau_1 :: s_1) &= D_1 \quad \ldots \quad (\tau_n :: s_n) = D_n & \Sigma (\kappa, c) = \bar{s}_n & \text{(constructor$_D$)} \\
(\kappa \tau_1 \ldots \tau_n :: c) &= c_\kappa D_n \\
((\alpha :: c_1 \cap \ldots \cap c_n \cap \ldots \cap c_q) :: c_n) &= \alpha_n & \text{(variable$_D$)} \\
((\alpha :: s) :: d) &= D & c \in \text{SUP} d & \pi_{d \to c} D & \text{(classrel$_D$)}
\end{align*}
\]
Rule classrelD only works on judgments of the form \( (\alpha :: s) :: c \), thus prohibiting pointless constructions followed by projections as in \( \pi_{d \rightarrow c} (d_k \ldots) \). There remains an ambiguity: there might be different paths \( c_1 \in \text{SUP} c_2, c_3 \in \text{SUP} c_1, \ldots, c_{n-1} \in \text{SUP} c_n \) in the class hierarchy from a subclass \( c_n \) to a superclass \( c_1 \). To make the system deterministic we assume a canonical path is chosen. The correctness proof below (implicitly) shows that the exact choice is immaterial.

The \( \llbracket \mathbf{t} \rrbracket \) function (relative to a context \( \Gamma \)) only affects type applications:

\[
\llbracket x :: \tau \rrbracket = x :: \tau \\
\llbracket \lambda x :: \tau . t \rrbracket = \lambda x :: \tau . \llbracket t \rrbracket \\
\llbracket t_1 \; t_2 \rrbracket = \llbracket t_1 \rrbracket \; \llbracket t_2 \rrbracket
\]

Note that \( \llbracket \mathbf{t} \rrbracket \) is injective, i.e. \( \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket \) implies \( t_1 = t_2 \). For succinctness we introduce two more abbreviations:

\[
\llbracket \alpha :: c_1 \cap \ldots \cap c_n \rrbracket = (\delta_{c_1} \alpha) \ldots (\delta_{c_n} \alpha) \\
\llbracket c_n [\pi_k] \rrbracket = c_n [\llbracket t_1 :: s_1 \rrbracket \ldots \llbracket t_k :: s_k \rrbracket] \text{ where } \Sigma(\kappa, c) = \pi_k
\]

The transformation shown in Table 1 maps each statement to a list of typeclass-free statements; \( \llbracket S \rrbracket \) is simply the concatenation of the transformation of each statement in \( S \). Transformed statements stemming from class and inst statements introduce dictionary constants \( c_n \) along with projections \( \pi_{d \rightarrow c} \) for subclass relations and \( f \) for constants associated with classes (class operations). See Table 2 for an example.

The transformation of inst reveals the role of coregularity for dictionary construction: each \( \llbracket d_{\kappa} [\alpha :: s] \rrbracket \) on the right hand side requires that the corre-

<table>
<thead>
<tr>
<th>statement</th>
<th>transformed statement(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fun } f :: \forall \alpha :: \pi_k :: \tau ) where ( f ; t_1 = t_1 \ldots f ; t_n = t_n )</td>
<td>( \text{fun } f :: \forall \pi_k \cdot (\alpha :: s) :: \pi_k \rightarrow \tau ) where ( f ; t_1 = t_1 \ldots f ; t_n = t_n )</td>
</tr>
<tr>
<td>data ( \kappa ; \pi_k = f_1 ) of ( \pi_k ) ( \ldots ) ( f_n ) of ( \pi_k )</td>
<td>data ( \kappa ; \pi_k = f_1 ) of ( \pi_k ) ( \ldots ) ( f_n ) of ( \pi_k )</td>
</tr>
<tr>
<td>( \text{class } c \subseteq c_1 \cap \ldots \cap c_m ) where ( f_1 :: \forall \alpha . \pi_1, \ldots, f_n :: \forall \alpha . \pi_n )</td>
<td>( \text{data } \delta_{c} \alpha = \Delta c ) of ( { \alpha :: c_1 \cap \ldots \cap c_m } ) ( \forall \alpha . \pi_1, \ldots, f_n :: \forall \alpha . \pi_n )</td>
</tr>
<tr>
<td>( \text{fun } \pi_{c \rightarrow c_1} :: \forall \alpha . \delta_{c} \alpha \rightarrow \delta_{c_1} \alpha ) where ( \pi_{c \rightarrow c_1} (\Delta c_1 ; c_1 \ldots c_m \pi_{c \rightarrow c_1} ; g_1 \ldots g_n) = c_1, 1 \leq i \leq m )</td>
<td>( \text{fun } f_1 :: \forall \alpha . \delta_{c} \alpha \rightarrow \tau_1 ) where ( f_1 (\Delta \pi_1, g_1 \ldots g_i \ldots g_n) = g_i, 1 \leq i \leq n )</td>
</tr>
<tr>
<td>( \text{inst } \kappa ; \pi_k :: c ) where ( f_1 = t_1, \ldots, f_n = t_n )</td>
<td>( \text{fun } c_n :: \forall \pi_k \cdot (\alpha :: s) :: \pi_k \rightarrow \delta_{c} (\kappa ; \pi_k) ) where ( { c_n [\alpha :: s] } = \Delta c_n [d_{n} :: [\pi_k :: s]] \ldots [d_{n} :: [\pi_k :: s]] ) ( \llbracket t \rrbracket_n )</td>
</tr>
</tbody>
</table>

Table 1. Dictionary construction for statements.
sponding sort constraints stemming from the inst statement that gave rise to the definition of \( d_k \) are as least as general as the \( \overline{\alpha} \sim \overline{s}_k \).

We assume distinct name spaces to embed dictionary constant names \( c_k \) and dictionary variable names \( \alpha_n \) into the name space of constants and term variables respectively. Each dictionary term constructors \( \Delta_k \) constructs a tuple containing the dictionaries of the direct superclasses of \( c \) together with the class operations declared in \( c \).

**Correctness**

Given a program \( S \) with \( S \sim (\Gamma, E) \) and its transformed counterpart \( (\bar{S}) \sim (\Gamma_D, E_D) \), we show how \( E \) and \( E_D \) are related. The main problem is that derivations in \( E_D \) contain symbols not present in \( E \). Thus intermediate terms in an \( E_D \)-derivation cannot always be viewed as \( \Gamma \)-terms. Hence we work with normal forms modulo certain projection rules, the rules stemming from class statements. For this fine-grained reasoning we move from equational logic to term rewriting. Instead of arbitrary equational proofs \((\Gamma_D, E_D) \vdash t_1 \equiv t_2 \) we consider rewrite proofs \( t_1 \xrightarrow{\Delta} t_2 \), see [8]. This models the evaluation of \( t_1 \). More precisely, we start with some \((\overline{s})\), reduce it to some \( \overline{t} \), see [8].

<table>
<thead>
<tr>
<th>Program</th>
<th>Transformed Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{data} Nat = Zero</td>
<td>\textbf{data} Nat = Zero</td>
</tr>
<tr>
<td>\textbf{data} List ( \alpha ) = Nil</td>
<td>\textbf{data} List ( \alpha ) = Nil</td>
</tr>
<tr>
<td>\textbf{class} Pls ( \alpha ) \textbf{where}</td>
<td>\textbf{class} Pls ( \alpha ) \textbf{where}</td>
</tr>
<tr>
<td>pls :: ( \forall \alpha \cdot \alpha \rightarrow \alpha \rightarrow \alpha )</td>
<td>pls :: ( \forall \alpha \cdot \alpha \rightarrow \alpha \rightarrow \alpha )</td>
</tr>
<tr>
<td>\textbf{class} Neutr ( \subseteq ) Pls ( \alpha ) \textbf{where}</td>
<td>\textbf{class} Neutr ( \subseteq ) Pls ( \alpha ) \textbf{where}</td>
</tr>
<tr>
<td>neutr :: ( \forall \alpha \cdot \alpha )</td>
<td>neutr :: ( \forall \alpha \cdot \alpha )</td>
</tr>
<tr>
<td>\textbf{fun} add :: Nat \rightarrow Nat \rightarrow Nat \textbf{where}</td>
<td>\textbf{fun} add :: Nat \rightarrow Nat \rightarrow Nat \textbf{where}</td>
</tr>
<tr>
<td></td>
<td>add Zero ( m ) = ( m )</td>
</tr>
<tr>
<td></td>
<td>add (Suc ( n )) ( m ) = Suc (add ( n ) ( m ))</td>
</tr>
<tr>
<td>\textbf{inst} Nat :: Pls ( \alpha ) \textbf{where}</td>
<td>\textbf{inst} Nat :: Pls ( \alpha ) \textbf{where}</td>
</tr>
<tr>
<td>pls = add</td>
<td>pls = add</td>
</tr>
<tr>
<td>\textbf{inst} Nat :: Neutr ( \alpha ) \textbf{where}</td>
<td>\textbf{inst} Nat :: Neutr ( \alpha ) \textbf{where}</td>
</tr>
<tr>
<td>neutr = Zero</td>
<td>neutr = Zero</td>
</tr>
<tr>
<td>\textbf{fun} sum :: ( \forall \alpha \cdot ) Neutr. List ( \alpha \rightarrow \alpha \rightarrow \alpha ) \textbf{where}</td>
<td>\textbf{fun} sum :: ( \forall \alpha \cdot ) Neutr. List ( \alpha \rightarrow \alpha \rightarrow \alpha ) \textbf{where}</td>
</tr>
<tr>
<td>( \sum \alpha \cdot \mathit{Nil} = \mathit{neutr} \left[ \alpha \right] )</td>
<td>( \sum \alpha \cdot \mathit{Nil} = \mathit{neutr} \left[ \alpha \right] )</td>
</tr>
<tr>
<td>( \sum \alpha \cdot ) (Cons ( \alpha \cdot ) ( x \cdot ) ( s \cdot )) = pls ( \left[ \alpha \right] \cdot ) (sum ( \left[ \alpha \right] \cdot ) ( x \cdot ) ( s \cdot ))</td>
<td>( \sum \alpha \cdot ) (Cons ( \alpha \cdot ) ( x \cdot ) ( s \cdot )) = pls ( \left[ \alpha \right] \cdot ) (sum ( \left[ \alpha \right] \cdot ) ( x \cdot ) ( s \cdot ))</td>
</tr>
<tr>
<td>\textbf{fun} val :: Nat \textbf{where}</td>
<td>\textbf{fun} val :: Nat \textbf{where}</td>
</tr>
<tr>
<td>( \mathit{val} = \mathit{sum} \left[ \mathit{Nat} \cdot \right] \cdot ) (Cons ( \mathit{Suc} \cdot ) ( \mathit{Zero} \cdot ) ( \mathit{Nil} \cdot ))</td>
<td>( \mathit{val} = \mathit{sum} \left[ \mathit{Neutr} \mathit{Nat} \cdot \right] \cdot ) (Cons ( \mathit{Suc} \cdot ) ( \mathit{Zero} \cdot ) ( \mathit{Nil} \cdot ))</td>
</tr>
</tbody>
</table>

\textbf{Table 2.} Dictionary construction applied to example program.
\[ s \rightarrow_E s' \], we want to conclude \( s \rightarrow_E s' \). The remainder of this section is dedicated to that proof.

Observe that (by definition of \( (S) \)) \( E_D \) can be partitioned as \( E_D = E_F \uplus E_I \uplus E_E \) where

- \( E_F \) denotes all equations stemming from transformed fun statements.
- \( E_I \) denotes all equations stemming from transformed inst statements that introduce a particular \( \Delta_c \): \( c_\kappa \overline{\pi} \equiv \Delta_c \).
- \( E_E \) denotes all equations stemming from transformed class statements that eliminate a particular \( \Delta_c \): \( p(\Delta_c \overline{\pi}) \equiv x_i \) where \( p \) is either a projection \( \pi_{c \rightarrow d} \) or a class operation \( f \).

Given an arbitrary reduction sequence \( t_1 \overset{*}{\rightarrow}_{E_D} t_2 \) where \( t_1 \) and \( t_2 \) are \( \Delta \)-free, we can assume w.l.o.g. that no reduction takes place underneath a \( \Delta \) — such reductions can always be postponed. This is because all arguments of \( \Delta \) on the left-hand side of any equation in \( E_D \) are (distinct) variables. Similarly, we can postpone all \( E_I \) steps up to the point where they are needed, i.e. in front of a corresponding \( E_E \) step. This allows to view derivations in a normal form where each \( E_I \) step occurs immediately before a corresponding \( E_E \) step. Now we collapse these \( E_I / E_E \) pairs into single \( E_{IE} \) steps defined as follows:

\[
E_{IE} = \{ p(c_\kappa \overline{\pi}) \equiv x_i \mid (c_\kappa \overline{\pi} \equiv \Delta_c \overline{\pi}) \in E_I \wedge (p(\Delta_c \overline{\pi}) \equiv x_i) \in E_E \} \]

Clearly, the equations in \( E_{IE} \) are consequences of those in \( E_I \) and \( E_E \). Neither \( E_F \) steps nor \( E_{IE} \) steps introduce \( \Delta s \), so any intermediate term is \( \Delta \)-free; hence no \( E_E \) steps remain. Thus we have transformed \( t_1 \overset{*}{\rightarrow}_{E_D} t_2 \) into \( t_1 \overset{*}{\rightarrow}_{E_F \uplus E_{IE}} t_2 \).

Again we partition our rule set: \( E_{IE} = E_{op} \uplus E_{\pi} \) where

- \( E_{op} \) contains equations for class operations: \( f(c_\kappa \overline{\pi}) \equiv \ldots \), and
- \( E_{\pi} \) contains equations for superclass projections: \( \pi_{c \rightarrow d}(c_\kappa \overline{\pi}) \equiv d_\kappa \ldots \).

This establishes a one-to-one correspondence between equations in \( E \) and equations in \( E_F \uplus E_{op} \) such that \( (t_1 \equiv t_2) \in E \) implies \( ([t_1] \equiv [t_2]) \in E_F \uplus E_{op} \).

For fun statements this holds by definition, for inst statements it holds by construction of \( E_{op} \). Finally we transform our \( E_F \uplus E_{IE} \) reduction sequence such that after each \( E_F \uplus E_{op} \) step we normalize w.r.t. \( E_{\pi} \). This transformation is accomplished by the following theorem:

**Theorem 5** Let \( R \) and \( P \) be two sets of defining equations in a context free of classes such that the left-hand sides of \( R \) and \( P \) do not overlap, \( P \) is confluent, terminating and right-linear, and the right-hand sides of \( P \) preserve \( \beta \)-normal forms (i.e. if a right-hand side of \( P \) is instantiated by \( \beta \)-normal forms, the result is in \( \beta \)-normal form). Then the following commutation property holds:

\[
t_1 \rightarrow_P t \rightarrow_R t_2 \text{ implies } \exists u. \ t_1 \overset{\ast}{\rightarrow}_R u \overset{\ast}{\rightarrow}_P t_2
\]

The proof is by a careful case distinction on the relative position of redexes.

By induction we obtain:
Corollary 6 If the assumptions of Theorem 5 hold, then \( t \xrightarrow{\ast} R; P \) \( t' \) implies \( t \downarrow_P (\sim_R \circ \sim_P)^* t' \downarrow_P \), where \( x \rightarrow y \) means \( x \xrightarrow{\ast} y \) and \( y \) is in normal form.

Setting \( R = E_F \uplus E_{op} \) and \( P = E_{\pi} \) we obtain: if \( \langle s \rangle \xrightarrow{\ast} E_F \uplus E_{op} \uplus E_{\pi} \) \( \langle s' \rangle \) then \( \langle s \rangle \xrightarrow{E_F \uplus E_{op} \uplus E_{\pi}} \langle s' \rangle \). Due to the form of the rules they satisfy the requirements of Theorem 5. In particular, \( E_F \uplus E_{op} \) and \( E_{\pi} \) are orthogonal because no \( \pi \) occurs in any left-hand side of \( E_F \) or \( E_{op} \). Also note that \( \langle s \rangle \downarrow_{E_{\pi}} = \langle s \rangle \) because \( \langle s \rangle \) contains no \( \pi \)-redexes.

Since each term \( t \) in a derivation \( \langle s \rangle \xrightarrow{E_F \uplus E_{op} \uplus E_{\pi}} \langle s' \rangle \) is \( \Delta \)-free and \( E_{\pi} \)-normalized, it is the image of an \( s'' \) such that \( \langle s'' \rangle = t \). Thus each \( E_F \uplus E_{op} \) step using an equation \( \langle s_1 \rangle \equiv \langle s_2 \rangle \) followed by \( E_{\pi} \)-normalization corresponds to an \( E \)-step using \( s_1 \equiv s_2 \). Normalization with \( E_{\pi} \) is necessary because substituting dictionaries into a translated term yields \( E_{\pi} \) redexes to access dictionaries of superclasses. In the original system \( E \) this is implicit due to subclassing.

Overall, we have now obtained the desired result:

Lemma 7 If \( \langle s \rangle \xrightarrow{E_D} \langle s' \rangle \) then \( s \xrightarrow{E} s' \).

6 Implementable systems

We will now discuss the step from a logical theory \((\Gamma, E)\) to a program. It can be seen as the inverse of \( \equiv \). More precisely:

Definition 8 A program \( S \) implements an equation system \((\Gamma, E)\) iff \( S \Rightarrow (\Gamma', E) \) such that \( \Gamma' \) is compatible with \( \Gamma \):

- \( TYP' \subseteq TYP \)
- \( SUP' \subseteq SUP \)
- \( \Sigma'(\kappa, c) \equiv \Sigma_{k} \) implies \( \exists \Sigma_{k}. \Sigma(\kappa, c) = \Sigma_{k} \land \forall 1 \leq i \leq k. \Gamma \vdash s'_i \subseteq s_i \)
- \( \Omega' = \forall x : s''_{k}. \tau \) implies \( \exists \Sigma_{k}. \Omega c = \forall x : s''_{k}. \tau \land \forall 1 \leq i \leq k. \Gamma \vdash s'_i \subseteq s_i \)

Compatibleness means that any expression which is valid with respect to \( \Gamma' \) is also valid with respect to \( \Gamma \). Permitting more restrictive sort constraints may be necessary for implementation reasons. For example, equality (=) is free of any class constraints in HOL but its implementation requires an equality class (§7).

Isabelle provides definition mechanisms (the details are immaterial) corresponding to our programming language statements \texttt{data}, \texttt{fun}, \texttt{class}, and \texttt{inst}. Systems \((\Gamma, E)\) defined purely in this manner are implementable. But there are other, more primitive ways to define functions in Isabelle which may not be directly implementable (see e.g. §4). To construct a program from \((\Gamma, E)\) we need certain extra-logical information not directly contained in \((\Gamma, E)\) anymore, e.g. what are the term constructors and the class operations. We will now isolate what is required and will then show that it enables us to assemble a program from a system \((\Gamma, E)\). We do not explicitly discuss the preprocessor which selects and transforms an initial set of equations into the required form (if possible).

Term constructors determine which constants must be introduced by \texttt{data} statements:
Definition 9 C is a set of term constructors for \((\Gamma, E)\) iff for each constant \(f\) occurring in the arguments of a defining equation \(f \in C\) holds, there are no defining equations in \(E\) for any \(f \in C\), and for each \(f \in C\) its generic type \(\Omega f\) is of the form \(\forall \alpha, \tau_n \rightarrow \kappa \alpha\), and each occurrence of \(f\) in the arguments of a defining equation is fully applied with \(n\) arguments.

Class and overloading discipline Defining equations may contain arbitrary type arguments whereas our programming language enforces a certain discipline. This is captured by an explicit association of constants to classes (corresponding to class statements):

Definition 10 An \(n\)-to-1 relation \(\in\) between constant symbols and class symbols is a class membership relation w.r.t. \(\Gamma\) iff for each \(f \in c\) its generic type \(\Omega f\) is of the form \(\forall \alpha :: c. \tau\), i.e. is generic in only one type argument of class \(c\). All constant symbols in the domain of \(\in\) are named class operations.

Given a set of defining equations \(E\), \((f \mid \kappa_k \ldots \equiv \ldots) \in E\) is called

- non-overloaded if \(\tau_m\) is of the form \([\alpha :: s_m]\) and all equations for \(f\) in \(E\) have the same type arguments.
- overloaded if \(m = 1\) and \(\tau_m\) is of the form \(\kappa \alpha :: s_k\), and there is no other \(\langle f [\kappa \alpha :: s_k] \ldots \equiv \ldots \rangle \in E\).

Non-overloaded definitions can be implemented by fun statements, overloaded definitions by inst statements. We now lift the definition of coregularity from contexts to systems of equations \(E\) in order to satisfy the coregularity requirement of the inst statement. \(E\) is coregular iff

\[
(f [\kappa \alpha :: s_m] \ldots \equiv \ldots) \in E, f \in c \text{ implies } \forall d, \Gamma \vdash c \subseteq d, g \in d.
\]

\[
\exists (g [\kappa \alpha :: s'_m] \ldots \equiv \ldots) \in E, \forall 1 \leq m. \Gamma \vdash s'_{n} \subseteq s_{n}
\]

In case \(c = d\), coregularity ensures that for a specific instance \((\kappa, c)\) all class operations are instantiated and occur with the same sort constraints.

Definition 11 A system \((\Gamma, E)\) obeys class discipline w.r.t. a class membership relation \(\in\) iff all defining equations in \(E\) are either overloaded or non-overloaded, and if for each \((f [\kappa \alpha :: s_k] \ldots \equiv \ldots) \in E\) there are \(c\) and \(s_k\) such that \(f \in c\), \(\Sigma(\kappa, c) = s_k\), and \(\Gamma \vdash s'_{i} \subseteq s_{i}\) for \(1 \leq i \leq k\). Furthermore \(E\) must be coregular.

In a system with term constructors \(C\) which obeys class discipline w.r.t. \(\in\), each constant symbol is either a term constructor (if \(f \in C\)), or a class operation (if \(f \in c\) for some \(c\)), or simply a function (otherwise). With this partitioning, \(E\) induces the more specialized context components \(\Omega\) and \(\Sigma:\)

Definition 12 Let \((\Gamma, E)\) be a system with term constructors \(C\) obeying class discipline w.r.t. \(\in\). Then \(\Omega f\) is identical to \(\Omega f\) unless \(f\) is a function, in which case \(\Omega f\) is derived from \(\Omega f = \forall \alpha :: s. \tau\) and any defining equation \(f [\alpha :: s] \ldots \equiv \ldots\) as follows: \(\Omega f = \forall \alpha :: s'. \tau\). For each defining equation \(f [\kappa \alpha :: s_k] \ldots \equiv \ldots\) with \(f \in c\) we define \(\Sigma(\kappa, c) = s_k\).
Given a program \( S \) with \( S \sim (I', E) \), by construction \( \tilde{\Omega} \) and \( \tilde{\Sigma} \) form the corresponding components of the context \( I' \).

**Definition 13** An expression \( f[\tau_n] \) occurs in a defining equation iff \( f[\tau_n] \) occurs in either the right hand side or in any argument.

Intuitively, this models a notion of “this function uses function \( f \)”.

Because the sort constraints in \( \tilde{\Omega} \) and \( \tilde{\Sigma} \) may be more restrictive than those in \( \Omega \) and \( \Sigma \), it is necessary to require well-sortedness of \( E \) explicitly:

**Definition 14** A system \((I,E)\) with term constructors \( C \) obeys class discipline w.r.t. \( \in \) is well-sorted iff for any constant expression \( f[\tau_n] \) occurring in any defining equation, \((SUP, TYP, \Sigma, \tilde{\Omega}) \vdash \tau_i :: s_i \) for \( 1 \leq i \leq n \), where \( \tilde{\Omega} f = \forall \alpha :: s_n, \tau \).

Note that well-sortedness can be achieved by the preprocessor which propagates the additional sort constraints through the system of equations.

**Dependency closedness** A requirement for programs \( S \) is that no statement relies on ingredients which have not been introduced so far; this induces a notion of dependencies of statements which is also reflected in the underlying system of defining equations. We describe this in terms of dependency graphs \( \rightarrow_E \) with directed edges where each node either refers to a term constructor, a generic class operation, a function (denoted by \( f \)) or refers to a particular instance of a class operation (denoted by \( f_\kappa \)). For brevity, we use the notation \( f_* \) referring uniformly to overloaded and non-overloaded constants, depending on which kind of defining equation \( f \) comes from.

**Definition 15** Given a well-sorted system \((I,E)\) with term constructors \( C \) obeying class discipline w.r.t. \( \in \), the dependency graph \( \rightarrow_E \) is defined by the following rules:

\[
\frac{(f[\tau] t \equiv t) \in E \quad g[\tau] \ occurs \ in \ (f[\tau] t \equiv t)}{f_* \rightarrow_E g} \quad \text{(dep)}
\]

\[
\frac{(f[\tau] t \equiv t) \in E \quad g[\tau] \ occurs \ in \ (f[\tau] t \equiv t) \quad \tilde{\Omega} g = \forall \alpha :: s, \tau}{c_\kappa \ occurs \ in \ ([\tau :: s])_{\Sigma} \quad h \in c \quad f_* \rightarrow_E h_\kappa} \quad \text{(dict)}
\]

\[
\frac{f \in c \quad g \in c \quad (\kappa, c) \in \text{dom } \Sigma \quad h \in c}{f_\kappa \rightarrow_E g_\kappa} \quad \text{(instop)}
\]

**dep** models dependencies of defining equations on class, data or fun statements.

**dict** models dependencies of defining equations on inst statements; the notation \( ([\tau :: s])_{\Sigma} \) means dictionary construction w.r.t. \( \Sigma \).

**instop** models that class operations which are members of the same class have to be instantiated simultaneously (by means of a corresponding inst statement).
Definition 16 A dependency graph $\rightarrow^E$ is closed iff

1. for each edge $f_* \rightarrow^E g_*$, $g_*$ is either a term constructor in $C$ or a generic class operation or a node with at least one outgoing edge;
2. the nodes of each strongly connected component of $\rightarrow^E$ consist either only of non-overloaded constants $f$ or of all overloaded constants $f_\kappa$, $f \in c$, for some fixed $\kappa$ and $c$.

The latter condition asserts that inst statements may not be mutually recursive, least of all in connection with fun statements.

Theorem 17 Each well-sorted system $(\Gamma, E)$ with term constructors $C$ obeying class discipline w.r.t. $\in$, where $\rightarrow^E$ is closed, is implementable.

Proof sketch Let $(\Gamma, E)$ have the required properties; a proof that $(\Gamma, E)$ is implementable consists of the following parts:

- Show that $C$ and $\in$ can be realized by data and class statements.
- Show that the defining equations $E$ are mappable to fun and inst statements, i.e. that there exists a partitioning of $E$ where each partition corresponds to a statement whose semantics is the original set of defining equations.
- Show that there exists an order of the above statements such that the corresponding list is a program.

The first is trivial by construction. For the second observe that class discipline yields partitions of non-overloaded defining equations corresponding to fun statements and overloaded defining equations corresponding to inst statements. The type and sort information needed for class, inst and fun statements comes from $\tilde{\Omega}$ and $\tilde{\Sigma}$. Well-sortedness of $(\Gamma, E)$ in the sense of Definition 14 guarantees that the resulting statements are typeable. The third requires some thought how to find out an appropriate order for a given list of statements. Essentially, any symbol $(c, \kappa$ or $f)$ must be introduced before it occurs in a statement. Furthermore all inst statements must occur before they are required for typing type applications $f[\tau]$. The following order guarantees this:

- data statements only depend on the existence of particular type constructors $\kappa$; so, data statements may be placed before any other kind of statements. data statements themselves are ordered such that any data statement depending on type constructor $\kappa$ is preceded by a data statement introducing $\kappa$; mutual dependencies result in mutual recursive data statements.
- class statements depend on the existence of type constructors $\kappa$ and superclasses $c$; since all $\kappa$ can be introduced by preceding data statements, only the classes have to be considered. Ordering class statements in topological order with respect to the subclass relation (starting with the top classes) results in an order where all superclasses of a class statement are introduced by preceding class statements.
The issue of \texttt{fun} and \texttt{inst} statements is more complicated because a \texttt{fun} statement may depend on an \texttt{inst} statement and also vice versa. But dependency closeness implies that the strongly connected components of $\rightarrow_{E}$ correspond exactly to (possibly mutually recursive) \texttt{fun} statements or \texttt{inst} statements; their order is determined by the graph. \hfill \Box

7 Executable equality

The constant denoting HOL equality ($=$) :: $\alpha \rightarrow \alpha \rightarrow \text{bool}$ is a purely logical construct. Its axiomatization is not in the form of defining equations. However, by providing an appropriate framework setup, we can derive defining equations for types which have an operational notion of equality (e.g. there is no operational equality on function types). Operational equality serves as example how the logic may be utilized to widen the possibilities for code generation without any need to extend the trusted code base of the framework.

When modeling operational equality we follow the Haskell approach: each equality type belongs to a type class \texttt{eq} ($=$) \in \texttt{eq}. Isabelle/HOL proves for each recursive datatype a set of defining equations for equality on that type, similar to Haskell’s “\texttt{deriving Eq}”:

\begin{itemize}
  \item Check whether all existing type constructors $\kappa'$ which $\kappa$ depends on are instances of \texttt{eq} ($\kappa' \ \overline{\text{eq}} :: \text{eq}$); if not, abort the whole procedure — then $\kappa$ does not support operational equality.
  \item Declare $\kappa$ an instance of \texttt{eq}, provided its type arguments are also instances of \texttt{eq}: $\kappa \overline{\text{eq}} :: \text{eq}$.
  \item Define a new constant \texttt{eq}$\kappa$[$\overline{\alpha}$ :: \texttt{eq}] \equiv (=)[$\kappa$[$\overline{\alpha}$ :: \texttt{eq}]].
  \item From this primitive definition prove injectiveness and distinctness equations as defining equations for \texttt{eq}$\kappa$:
    \begin{align*}
    \texttt{eq}_\kappa(f_i \overline{x_m}) &= (=) \overline{x_1} \land \ldots \land (=) \overline{x_m} \overline{y_m} \\
    \texttt{eq}_\kappa(f_i \ldots)(f_j \ldots) &= \text{False}, \text{ for } i \neq j
    \end{align*}
    where empty conjunctions collapse to True and recursive calls of equality
    \begin{align*}
    (=)[\kappa \overline{\alpha} :: \texttt{eq}] & \equiv \texttt{eq}_\kappa, \text{as defining equation for}
    (=)[\kappa \overline{\alpha} :: \texttt{eq}].
    \end{align*}
  \item Use the symmetric definition $(=)[\kappa \overline{\alpha} :: \texttt{eq}] \equiv \texttt{eq}_\kappa$ as defining equation for
    $(=)[\kappa \overline{\alpha} :: \texttt{eq}]$.
\end{itemize}

On code generation, the preprocessor (§2) propagates \texttt{eq} sort constraints through the system of defining equations, e.g. a defining equation $(\neq)[\alpha] \ x \ y \equiv \neg(=)[\alpha \ \overline{x} \ \overline{y})$ is constrained to $(\neq)[\alpha :: \texttt{eq}] \ x \ y \equiv \neg(=)[\alpha :: \texttt{eq}] \ x \ y)$. Because this is a purely logical inference, the translation process is completely unchanged. Thus, operational equality is treated inside the logic. In particular, this approach is completely independent from any target-language specific notion of equality (e.g. SML’s polymorphic $(\text{op} =)$).

8 Conclusion

We have presented the design of Isabelle/HOL’s latest code generator (available
in the development snapshot) which is, for the first time, able to deal with type classes. The correctness of code generation, in particular the relationship between type classes and dictionaries, is established by proof theoretic means. The trusted code base is minimized by working with a conceptually simple programming language with a straightforward equational semantics.

References


CSP Revisited

Florian Kammlüßer

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract. In this paper we revisit the formalization of Communicating Sequential Processes (CSP) [2] in Isabelle/HOL. We offer a simple alternative embedding of this specification language for distributed processes that makes use of as many standard features of the underlying Higher Order Logic of Isabelle, like datatypes and the formalization of fixpoints due to Tarski.

1 Introduction

The specification language CSP for Communicating Sequential Processes is a classical tool for specifying and reasoning about parallel communicating processes.

Although there exists the FDR tool for the analysis of CSP specifications there has always been a considerable amount of parallel activities to mechanize the CSP language in HOL systems. The first formalization of this theory in Isabelle/HOL [4] dates back to the early nineties [1]. Later the work by Tej and Wolff contributed a fully fledged support tool. Yet there has been ongoing research on formalizing CSP in Isabelle/HOL [3].

The reason to offer yet another formalization is twofold. Firstly, the earlier formalization by Wolff has been frozen in with Isabelle version 98 because the formalization was quite closely intertwined with the Isabelle source code. Thus it became more and more difficult for the developers to adjust their CSP tool repeatedly to newer Isabelle versions. We believe that it must have been for this particular reason that more recently a new project was initiated where the authors reformalized the CSP calculus. However, contemplating this more recent formalization we found that it uses sometimes contrived implementations for the definition of recursive processes relying implicitly on Tarski's fixpoint theory. However, they do not use this theory although it is available in Isabelle/HOL (theory Fixedpoint.thy). There might be pragmatic reasons not to build on it but we find that this unnecessarily endangers consistency. As we also use CSP in teaching, we have a particular interest to show the concepts of CSP in a transparent light while offering students a tool environment.

For these reasons we construct CSP again in Isabelle/HOL while keeping the formalization as lightweight and practical as possible. Therefore we aimed at using where possible the powerful datatype package in combination with primitive recursive definitions of the operators and reusing as much as possible existing predefined theories like the fixpoint theory provided in Isabelle/HOL.
In this paper we present first the definition of the CSP syntax using a simple
datatype definition in Section 2. Then we define the semantics of CSP in Section
3. In this section we also briefly summarize the two extensions of the simple trace
semantics to failures and divergences. For the semantics of recursive processes
we use the existing theory of fixpoints in Isabelle/HOL. In Section 4 we consider
the notion of process refinement that is central to CSP for stepwise development
of specifications. Finally, in Section 5 we compare our approach in particular to
the earlier formalization by Wolff and draw some general conclusions.

2 The Syntax of CSP Processes

The first most decisive design decision in any CSP specification is the definition
of the possible communication events $\Sigma$. A process in CSP is then completely
defined by the events it can communicate. There is one basic process called STOP
that does not communicate anything. It represents the deadlock process. Other
processes may be constructed from existing processes and events. To build a
process that communicates sequentially the prefixing operator $\rightarrow$ is used. Given
an event $a \in \Sigma$ and a process $P$ the constructed process $a \rightarrow P$ communicates
first the event $a$ and after that it behaves like process $P$. Given two processes
$P$ and $Q$ we can construct the deterministic choice $P \Box Q$ and the nondetermin-
istic choice $P \sqcap Q$ between the two processes. Parallel composition of processes
is also possible. There are four possibilities for parallel composition. The inter-
leave operator combines two processes $P$ and $Q$ in such a way that the resulting
process $P \parallel| Q$ enables all possible interleavings of the events communicated by
either of the two. By contrast, in the parallel composition $P \parallel Q$ the resulting
process communicate only events that can be communicated by either of the two
processes simultaneously. That is in the interleaving there is no synchronization
between the communication of $P$ and $Q$. For a more precise control over inde-
pendence of communication on the one side and synchronization on the other
side there are the operators generalized and alphabetized parallel (here genpar
and alphpar). In the generalized parallel operator the set $A$ of events over which
the processes $P$ and $Q$ must agree is parameterized. That is in genpar $P A Q$ the
processes $P$ and $Q$ must synchronize over all events in $A$ but can communicate
independently all other events outside $A$. The hiding operator finally enables to
hide selected events from the visible part of a communication.

The following datatype definition defines the syntax of CSP processes.

```plaintext
datatype $\alpha$ process =
  STOP
| prefix $\alpha$ ($\alpha$ process)  (infixl "\rightarrow" 50)
| dchoice ($\alpha$ process) ($\alpha$ process) (infixl "\Box" 50)
| nchoice ($\alpha$ process) ($\alpha$ process) (infixl "\sqcap" 50)
| parallel ($\alpha$ process) ($\alpha$ process) (infixl "\parallel" 50)
| interleave ($\alpha$ process) ($\alpha$ process) (infixl "\parallel|" 50)
| genpar ($\alpha$ process) ($\alpha$ set) ($\alpha$ process)
| hiding ($\alpha$ process) ($\alpha$ set) (infixl "\backslash" 50)
```

CSP Revisited 145
3 Semantics

The first semantical model of CSP is the trace model. There are also the failures and the failures and divergences model that are refined models for CSP processes, but all build on the traces model.

3.1 Trace Model

For an alphabet $\Sigma$ the set $\Sigma^*$ denotes the set of all finite concatenations of elements of $\Sigma$ including the empty element. A word in $\Sigma^*$ is a concatenation $(a_1, a_2, \ldots)$ for elements $a_1, a_2, \ldots \in \Sigma$. The most natural presentation of traces in Isabelle/HOL is given by the list datatype, a well supported theory of Isabelle/HOL.

types $\alpha \text{ trace} = \alpha \text{ list}$

In order to define the first simple trace semantics of processes we use that classical filter operator on traces annotated as $\|\cdot\|_*$. 

consts

filter :: $[\alpha \text{ trace}, \alpha \text{ set}] \Rightarrow \alpha \text{ trace}$ (infixl "\|\cdot\|" 50)

Given a trace $s$ and a set of events $A$ it returns the trace that is obtained from $s$ by deleting all events that are not in $A$. This semantics can be given by the following two primitive recursive definitions. Note, that we reuse here the empty list $[]$ to represent the empty trace: the above types definition of traces keeps the underlying list syntax available for traces. Similarly, we reuse $\#$, the list constructor, as concatenation for traces.$^1$

primrec

filter_nil : "$[] \| A = []$"
filter_cons: "$(a \# l) \| A = (\text{if } a : A \text{ then } a \# (l \| A) \text{ else } (l \| A))$"

Next we declare the following constant for the trace semantics of CSP processes.

traces :: $\alpha \text{ process} \Rightarrow (\alpha \text{ trace}) \text{ set}$ ("[\_ \_]" [21] 20)

The idea of this first coarse semantics of CSP is to assign to each process the set of traces that record the possible behaviour of this process at each moment in time. That is, the empty trace is always an element of the traces of any process as it represents the moment when nothing yet has been communicated. It is also implied that the traces of any process are prefixed-closed: if a trace $t$ is in the set of traces($P$), then all prefixes of $t$ must be contained in traces($P$).

The traces of a process are now defined by underlying each of the possible constructors with a semantics. It is worth noting at this point that we can formulate these semantical rules again by primitive recursion. In the following we explain the pieces of a whole primrec section line by line at the same time.

$^1$ We may easily introduce syntactic sugar to cover the original list notions with the CSP specific syntax.
sheding some light on the trace semantics. The STOP process’ semantics is that it does never communicate anything; yet only the empty trace describes this behaviour.

\[
\text{primrec}
\text{traces_STOP} : \begin{bmatrix}
\text{STOP}
\end{bmatrix} = \{ \langle \rangle \}
\]

The traces for prefixing a process \(P\) by \(a\) are built by juxtaposing the \(a\) in front of all traces of \(P\); the empty trace has to be inserted afterwards again as it is lost in this procedure.

\[
\text{traces_prefix}:
\begin{bmatrix}
a \rightarrow P
\end{bmatrix} = \{ \langle \rangle \} \cup \{ t \mid \exists t'. \ t = a \# t' \wedge t' \in \begin{bmatrix} P \end{bmatrix} \}
\]

The deterministic choice between two processes \(P\) and \(Q\) opens up two branches in the behaviour of the process: hence the traces of the two individual processes are unified.

\[
\text{traces_dchoice} : \begin{bmatrix}
P \Box Q
\end{bmatrix} = \begin{bmatrix} P \end{bmatrix} \cup \begin{bmatrix} Q \end{bmatrix}
\]

The nondeterministic choice is meant to be a choice which is not influenced by the environment, but is internally taken (by the system). However, considering the traces, which build nothing more than a transcript of possible behaviours, the way the choice has been taken at a certain point is lost: the traces for the nondeterministic choice are identical to the traces of the deterministic choice.

\[
\text{traces_nchoice} : \begin{bmatrix}
P \cap Q
\end{bmatrix} = \begin{bmatrix} P \end{bmatrix} \cup \begin{bmatrix} Q \end{bmatrix}
\]

This shortcoming of the trace semantics leads to the extension of the semantical model by failures (see below).

The four different forms of parallel operators are defined as follows. In general the processes have to agree on each event they communicate: their traces must be shared, i.e. the traces of the resulting parallel composition is the intersection of the traces of each individual process.

\[
\text{traces_parallel} : \begin{bmatrix}
P \parallel Q
\end{bmatrix} = \begin{bmatrix} P \end{bmatrix} \cap \begin{bmatrix} Q \end{bmatrix}
\]

In the other extreme, the interleaving of two processes that are not synchronized at all, each of the combined processes has its own contribution to a trace of the combination. Technically we collect those combined traces \(s\) as restrictions to individual sets of events traces for each of the constituent processes such that these individual sets unify to the set of all traces that occur in \(s\).

\[
\text{traces_inter} : \begin{bmatrix}
P \parallel| Q
\end{bmatrix} = \{ s \mid \exists X Y . s \mid X \in \begin{bmatrix} P \end{bmatrix} \wedge s \mid Y \in \begin{bmatrix} Q \end{bmatrix} \wedge \text{set } s = X \cup Y \}
\]

The intermediate solution between the two – some portion \(A\) of events needs to be synchronized – is expressed in a similar manner. The restriction set of events of each of the processes \(P\) and \(Q\) includes here the set \(A\) thus enforcing that the events of \(A\) are communicated by both.
traces_genpar : "[[ genpar P A Q ]] = 
{ s . \exists X Y . s \restriction ((A \cup X) \in [[ P ]] \land s \restriction ((A \cup Y) \in [[ Q ]])
\land \text{set } s = X \cup A \cup Y )"

Hiding a set of events \( A \) in a process \( P \) simply means that in all traces of the original process \( P \) each event that is in \( A \) has to be eliminated. This can be encoded in the language of Isabelle/HOL by mapping a function that filters the complement of \( A \) with respect to \( \Sigma \) out of any trace \( s \) over the set of all traces of \( P \). Here \(-\) is the symmetric set difference and \( ' \) the operator building the image of a function applied to a set.

traces_hide: "[[ P \setminus A ]] = (\lambda s . s \restriction (\Sigma - A))'(\{ [[ P ]]\)"

3.2 Recursion

To construct non-trivial processes the use of recursion is enabled. Users of CSP may casually write recursive equation for the definition of processes, like the following process communicating sequences of \( a \)'s of arbitrary length.

\[ P = a \rightarrow P \]

Semantically such a recursive equation is resolved in the classical way as the the fixpoint of a corresponding functorial. For the example, the fixpoint of the functorial \( \lambda s.a \rightarrow s \) is chosen. Now, as we have defined the semantics of our CSP processes as sets (of traces) we have to consider functorials over such sets. As powersets with the subset relation form complete partially ordered sets we can use the theory of fixpoints in Isabelle/HOL to assign a meaning to recursive functions.

Now an interesting problem arises. As we base our embedding on a syntactical characterization of CSP operators by a datatype a recursive rescription is also syntactical. However, the fixpoints are only defined in the semantics, because it is there that we arrive at a function representing the meaning (syntactically the recursion leads to ever longer nested calls). In order to achieve this we define the set of all trace sets that are ever reached by any process over an alphabet \( \Sigma \).

\[ \text{constdefs} \]
\[ \text{Traces :: } ((\alpha \text{ trace}) \text{ set}) \text{ set} \]
\[ "\text{Traces == } \{ A :: (\alpha \text{ trace}) \text{ set}. \exists P . \text{ traces } P = A \}" \]

We define a higher order function that transforms a functorial over syntactic processes over a functorial over traces, i.e. in the semantics. We need to reconstruct the syntactical process that has a certain semantics in order to define the latter semantical factorial. To this end, we use the Hilbert operator, which enables the selection of some syntactical process that is mapped to a given trace set \( y \).

\[ \text{recursion_prep :: } (\alpha \text{ process } \Rightarrow \alpha \text{ process}) \Rightarrow ((\alpha \text{ trace})\text{set } \Rightarrow (\alpha \text{ trace})\text{set}) \]
\[ "\text{recursion_prep } f == (\lambda y :: (\alpha \text{ trace})\text{set } . \text{ if } y \in \text{Traces then } [[ f (\text{SOME } z . \text{ traces } z = y)] \text{ else } {}})" \]
To reassure ourselves that this transformation function works we prove the following lemma.

\[ y \in \text{Traces} \implies \exists \ z. \ \text{traces} \ z = y \]

CSP uses the notation \( \mu \) for the least fixed point operator. Given the transformation function \( \text{recursion\_prep} \) we can use the predefined least fixed point operator to assign a semantics to recursive process definitions.

\[
\text{constsdefs} \\
\mu :: \ ((\alpha \ \text{process} \Rightarrow \alpha \ \text{process}) \Rightarrow (\alpha \ \text{trace}) \ \text{set}) \\
\mu f = \text{lfp} (\text{recursion\_prep} f)
\]

### 3.3 Divergences

Through hiding in combination with recursion arises another problem in the behavioural specification of processes. Consider the following recursive process.

\[ P = (a \rightarrow P) \setminus a \]

Process \( P \) does not communicate anything visible, as the only event \( a \) that it communicates is hidden. This process behaves like the process \( \text{STOP} \) when considering just its traces.\(^2\) However, in difference to \( \text{STOP} \) which does not communicate at all \( P \) has an endless invisible activity of communication going on.

The behaviour expressed by process \( P \) is considered as a divergence in CSP. A process that may behave at a certain point like the above process is considered as nonterminating. To differentiate this behaviour from a process like \( \text{STOP} \) that does not do anything, divergences are explicitly introduced as the third layer of semantics in CSP.

### 3.4 Failures and Divergences

As we have seen in the previous sections there are two phenomena not expressible in the simple trace model of CSP.

- deterministic and nondeterministic choice are not distinguishable
- deadlock and livelock are not distinguishable.

For the former reason CSP considers a second extension of its semantical model by sets of events that may be refused at each moment during a process. These so-called refusal sets combined with a trace representing the particular moment when this refusal is possible form so-called failures. The latter reason gives rise to the extension of the semantical model by a third component: the divergences. These are sets of traces after which a process can behave like the above in that it produces an infinite invisible activity.

\(^2\) and also failures as we will see later
Formally, we define first divergences because failures may be defined reusing divergences. The divergences of a process are a set of traces that record the traces leading up to a divergence point and thereafter admit arbitrary behaviour.\(^3\)

\textbf{consts}

\begin{align*}
\text{D} &:: \alpha \text{ process} \Rightarrow (\alpha \text{ trace}) \text{ set} \\
\end{align*}

The divergences of STOP can now exhibit that this process is really a livelock, i.e. does not diverge. The case of the prefixing operator shows the basic idea of divergences: divergences record the behaviour of the process up to the point of divergence, thereafter any behaviour is possible in case of divergence. Similarly for the two choice operators alike a possible divergence of one of the combined processes renders the choice divergent. For the parallel operators we give only the more general parallel operator, as the others may be expressed in terms of this.

\textbf{primrec}

\begin{align*}
\text{Divergences}_\text{STOP} &:: "D(\text{STOP}) = \{}" \\
\text{Divergences}_\text{prefix} &:: "D (a \rightarrow P) = \{ s \mid \exists t. s = a \# t \land t \in D(P) \}" \\
\text{Divergences}_\text{dchoice} &:: "D(P \sqcap Q) = (D P) \cup (D Q)" \\
\text{Divergences}_\text{nchoice} &:: "D(P \sqcup Q) = (D P) \cup (D Q)" \\
\text{Divergences}_\text{genpar} &:: "D (\text{genpar} P A Q) = \\
& \{ s \mid \exists X Y. (s \mid (A \cup X) \in [P] \land s \in D(Q)) \lor \ (s \mid (A \cup Y) \in [Q] \land s \in D(P)) \}" \\
\text{Divergences}_\text{hide} &:: "D(P \setminus A) = \{ s \mid \exists t. t \in D(P) \land s = t \mid (\Sigma - A)\}"
\end{align*}

Failures are pairs of traces and sets of events – the refusals – that a process can refuse to communicate at the point described by the trace.

\textbf{F} :: \alpha \text{ process} \Rightarrow (\alpha \text{ trace} \times \alpha \text{ set}) \text{ set}

The failures of STOP reflect that at the empty trace this process can refuse any set of possible events. The failures of a process \(P\) prefixed by event \(a\) are characterized as follows: any event other than \(a\) any be refused initially – at the empty trace – afterwards anything that may be refused by \(P\). The failures of the nondeterministic choice are now just the union of the failures of each of the combined processes. This definition reflects that an internal choice may refuse a behaviour even if it is for choice. By contrast the deterministic – or external – choice is now differing in the failures model: initially it may only refuse if both processes may refuse, but all further refusals of both processes are then possible. In addition it may refuse anything if it can initially already diverge. For hiding the failures are derived by referring to the failures of the original process adding the hidden events in the refusals and in case of divergence again any failure.

\textbf{primrec}

\begin{align*}
\text{Failures}_\text{STOP} &:: "F (\text{STOP}) = \{} (s,X) \mid s = \[] \land X \subseteq \Sigma \}"
\end{align*}

\(^3\)In CSP the view is taken that a process that may diverge at some point does not convey any sensible behaviour. Therefore it is assigned any possible behaviour from that point onwards.
Failures_prefix: "F (a → P) = \{ (s,X) | s = [] \land a \notin X \} \cup
\{ (s, X) | \exists s'. s = a # s' \land (s',X) \in F(P) \}"

Failures_dchoice: "F(P □ Q) = (F P) \cup (F Q)"

Failures_nchoice: "F(P \otimes Q) = \{(s,X) | s = [] \land (s,X) \in (F P) \cap (F Q) \}
\cup \{(s,X) | s \neq [] \land (s,X) \in (F P) \cup (F Q) \}
\cup \{ (s,X) | s = [] \land [] \in D(P) \cup D(Q) \}"

Failures_hide: "F(P \setminus A) = \{(s,X) | \exists t. s = t \upharpoonright (\Sigma - A) \land (t, X \cup A) \in F(P) \} \cup
\{(s,X) | s \in D(P \setminus A) \}"

4 Refinement

Refinement of processes is defined on all three levels of the semantical model of CSP. In the earlier papers [1, 5] the authors use the so-called process ordering which is coarser than the classical ordering given by subset relations on the denotations. Tej and Wolff argue that in their mechanization they need this ordering as they need to cope with unbounded nondeterminism. For our principal investigation of the reformalization of CSP using datatype and primitive recursion we stick to the simpler classical refinement ordering.

Process refinement is firstly an important tool for a stepwise development of specifications in CSP: it guarantees that a process that is a refinement behaves inside the behavioural specification of the more abstract process. On the other hand it is also used to abstractly express the behaviour of processes thereby motivating the role of specification versus implementation.

The basic idea is simple. A more abstract process is a more general description of the allowed behaviour. Thus, for the most intuitive, the trace model, it is clear that the semantics of an abstract process contains more possible behaviour than its refinement. To put it the other way round: an implementation (or refinement) of a specification may pick out some of the behaviour that is granted as allowed behaviour by its specification. Hence, the refinement \(\subseteq_T\) on the trace semantics of processes is simply given by the \textit{subset relation} on the semantical denotations of two processes. The reading of \(P \subseteq_T Q\) is however somewhat unnaturally defined as \(P\) is refined by \(Q\) thereby inversing the direction of the \(\subseteq\).

constdefs
\(\subseteq_T:: [\alpha \ \text{process}, \ \alpha \ \text{process}] \Rightarrow \text{bool} \ (\text{infixl} \ 50)\)
"P \subseteq_T Q = \{ | Q \subseteq | P | \}"

For failures, a similar idea applies. Failures contain the traces of the trace model in their first component. Hence it is clear that here the same subsumption relation should hold. With respect to the refusals possible at each point the subsumption does in fact hold in the same sense: if the abstract process may at some point – after a possible trace – refuse certain sets of events, then its implementation may chose to refuse a subset of these possible refusal sets. This concept realizes the idea that the abstract process is also more general in the \textit{must} behaviour. The implementation can only be less chosy. Hence, the failures refinement is also simply given by subsumption of the failures of the two processes.
Finally, for the divergences the reasoning is more intricate. Divergence is seen as indeterminate behaviour. If a process may diverge at a certain point this is interpreted in the CSP world as undefined or unspecified behaviour. Therefore it turns out nicely here that in the divergence model processes have been assigned all possible behaviours up from possible points of divergence. Hence a process that may diverge at a point given by trace \( t \) is always more abstract than a process that has identical behaviour up to that point \( t \) but does not diverge then, i.e. has just a subset of all combinatorically possible trace continuations. Again the divergence refinement is the subset relation of the divergences subsuming the failures refinement.

\[
\subseteq_{FD} :: [\alpha \text{ process}, \alpha \text{ process}] \Rightarrow \text{bool} \quad \text{(infixl 50)}
\]

\[
P \subseteq_{FD} Q = FQ \subseteq FP \land DQ \subseteq DP
\]

5 Discussion

We first discuss the differences to the earlier formalization [5] of CSP in Isabelle/HOL before we draw some general conclusions.

5.1 Comparison

In brief, our formalization works the other way round than the formalization of Tej and Wolff. They use the wellformedness rules of CSP as a big conjunction to define a type of processes in a type definition. Then they define the properties of the operators using the abstraction and representation functions given by the type definition. This is a rather technical process of transferring properties from one domain to another. In our formalization using the datatype definition, we just go the latter step. We use the datatype feature of Isabelle and the list database to build up a wellformed process definition. Then we can use the same definitions they use inside the abstraction functions to define the semantical annotations as functions over the datatype of process. Our approach saves some work by using the now well established features of Isabelle/HOL.

We also use the predefined fixedpoint theory of Isabelle/HOL to define the semantics of recursive processes. Although there we have to twist our model in a slightly unnatural way, we arrive at using the given fixedpoint constructor because our semantical model is based on sets whereas Tej and Wolff use a self defined type as their semantics. Therefore they have to recreate the entire Tarski fixpoint theory using axiomatic type classes in order to build an instantiation to their process type.
5.2 Conclusions

Compared to the very well elaborated theory of CSP given by Tej and Wolff our formalization is just a first experimental sketch to test feasibility of this approach. We believe it is important to keep a formalization lightweight and as simple as possible. The historical development shows that it is well worth trying to keep things simple. Although initially Isabelle/HOL has been intended to be a tool for such deep integrated instantiations as the HOL-CSP tool by Tej and Wolff, it seems nowadays that the applications must not be dependent too much on the inner workings of the implementations. Otherwise, decisive changes in the development of Isabelle/HOL make it impossible to keep such instantiations up to date.

References

Acyclicity and Finite Linear Extendability: a Formal and Constructive Equivalence

Stéphane Le Roux*

LIP, École normale supérieure de Lyon, France
stephane.le.roux@ens-lyon.fr

Abstract. Linear extension of partial orders was addressed in the late 1920's. Its computer-oriented version, i.e., topological sorting of finite partial orders, arose in the late 1950's. However, those issues have not yet been considered from a viewpoint both formal and constructive; this paper discusses a few related claims formally proved with the constructive proof assistant Coq. For example it states that a given decidable binary relation is acyclic and equality is decidable on its domain iff an irreflexive linear extension can be computed uniformly for any of its finite restriction.

Keywords: Binary relation, finite restriction, linear extension, (non-)uniform computability, topological sorting, constructivism, induction, proof assistant.

1 Introduction

This paper is a shortened version of the research report [8]. The report is readable by a mathematician unfamiliar with constructive and computability issues and Coq formalism; this paper assumes basic knowledge in those fields.

1.1 Transitive Closure, Linear Extension, and Topological Sorting

The calculus of binary relations was developed by De Morgan around 1860. The notion of transitive closure of a binary relation (smallest transitive binary relation including a given binary relation) was defined in different manners by different people about 1890. See Pratt [9] for a historical account. In 1930, Szpilrajn [10] proved that, assuming the axiom of choice, any partial order has a linear extension, i.e., is included in some total order. The proof invokes a notion close to transitive closure. Szpilrajn acknowledged that Banach, Kuratowsky, and Tarski had found unpublished proofs of the same result. In the late 1950’s, The US Navy [2] designed PERT (Program Evaluation Research Task or Project Evaluation Review Techniques) for management and scheduling purpose. This tool partly consists in splitting a big project into small jobs on a chart and expressing with arrows when one job has to be done before another one can start.

* This research was partly supported by Collège Doctoral Franco-Japonais
up. In order to study the resulting directed graph, Jarnagin [4] introduced a finite and algorithmic version of Szpilrajn’s result. This gave birth to the widely studied topological sorting issue, which spread to the industry early 1960’s (see [7] and [5]). Some technical details and computer-oriented examples can be found in Knuth’s book [6].

1.2 Contribution

This paper revisits a few folklore results involving transitive closure, excluded middle, computability, linear extension, and topological sorting. Most of the properties are logical equivalences instead of one-way implications, which suggests maximal generality. Claims have been fully formalized (and proved) in Coq and then slightly modified in order to fit in the Coq-related CoLoR library [3].

In this paper, a binary relation over an arbitrary set is said to be middle-excluding if for any two elements in the set, either they are related or they are not. The intermediate result of this paper implies that in an arbitrary set with decidable (resp. middle-excluding) equality, a binary relation is decidable (resp. middle-excluding) if the transitive closures of its finite restrictions are uniformly decidable (resp. middle-excluding). The main result splits into two parts, one on excluded middle and one on computability: First, consider a middle-excluding relation. It is acyclic and equality on its domain is middle-excluding if its restriction to any finite set has a middle-excluding irreflexive linear extension. Second, consider \( R \) a decidable binary relation over \( A \). The following three propositions are equivalent. Note that computability of linear extensions is non-uniform in the second proposition but uniform in the third one.

- Equality on \( A \) is decidable and \( R \) is acyclic.
- Equality on \( A \) is decidable and every finite restriction of \( R \) has a decidable linear extension.
- There exists a computable function that waits for finite restrictions of \( R \) and returns (decidable) linear extensions of them.

This paper follows the structure of the underlying Coq development but some straightforward results are omitted. Proofs are written in plain English. The main result in this paper relies on an intermediate one, and is itself invoked in a game theoretic proof (in Coq) not published yet.

1.3 Contents

Some basic objects from the Coq Library [1] are not defined in this paper. Section 2 talks about excluded middle and decidability and section 3 about lists. Section 4 discusses the notion of transitive closure, irreflexivity, and finite restrictions of a binary relation. Section 5 defines paths with respect to a binary relation and proves their correspondence with transitive closure. It also defines bounded paths that are proved to preserve decidability and middle-exclusion properties of the original relation. Since bounded paths and paths are by some
means equivalent on finite sets, subsection 5.4 states the intermediate result. Subsection 6.1 defines relation totality over finite sets. Subsections 6.2 to 6.5 define an acyclicity-preserving conditional single-arc addition (to a relation), and an acyclicity-preserving multi-stage arc addition over finite sets, which consists in repeating in turn single-arc addition and transitive closure. This procedure helps state equivalences for linear extension in 6.6 and topological sorting in 6.7.

1.4 Convention

Let $A$ be a $Set$. Throughout this paper $x, y, z,$ and $t$ implicitly refer to objects of type $A$. In the same way $R, R'$, and $R''$ refer to binary relations over $A; l, l', and l''$ to lists over $A$, and $n$ to natural numbers. For the sake of readability, types will sometimes be omitted according to the above convention, even in formal statements where Coq could not infer them. The notation $\neg P$ stands for $P \rightarrow False$, $x \neq y$ for $x=y \rightarrow False$, and $\exists x, P$ for $(\exists x, P) \rightarrow False$.

2 On Excluded Middle and Decidability

The following two definitions respectively say that equality on $A$ is middle-excluding and that a given binary relation over $A$ is middle-excluding.

Definition $eq\_midex := \forall x, y, x=y \lor x\neq y$.

Definition $rel\_midex R := \forall x, y, R x y \lor \neg R x y$.

The next two definitions respectively say that equality on $A$ is decidable and that a given binary relation over $A$ is decidable.

Definition $eq\_dec := \forall x, y, \{x=y\}+\{x\neq y\}$.

Definition $rel\_dec R := \forall x, y, \{R x y\}+\{\neg R x y\}$.

The following two lemmas justify the representation of decidability used in this paper.

Lemma $rel\_dec\_bool : \forall R,$

$rel\_dec R \rightarrow \{f : A \rightarrow A \rightarrow bool \mid \forall x y : A, if f x y then R x y else \neg R x y\}.$

Lemma $bool\_rel\_dec : \forall R,$

$\{f : A \rightarrow A \rightarrow bool \mid \forall x y : A, if f x y then R x y else \neg R x y\} \rightarrow rel\_dec R.$

“Decidability implies excluded middle”, as shown below.

Lemma $eq\_dec\_midex : eq\_dec \rightarrow eq\_midex$.

Lemma $rel\_dec\_midex : rel\_dec \rightarrow rel\_midex$.

3 On Lists

If equality is middle-excluding on $A$ and if an element occurs in a list built over $A$, then the list can be decomposed into three parts: a list, one occurrence of the element, and a second list where the element does not occur.
Lemma \( \text{Inelim\_right} : \text{eq\_midex} \rightarrow \forall \, x \, \ell, \)
\( \text{In} \, x \, \ell \rightarrow \exists \, \ell' \, \exists \, \ell'' \, \ell = \ell' + + (x :: \ell'') \land \neg \text{In} \, x \, \ell''. \)

Proof By induction on \( \ell. \) For the inductive case, case split on \( x \) occurring in the head or the tail of \( \ell. \)

The predicate \( \text{repeat\_free} \) says that no element occurs more than once in a given list. It is defined by recursion on its sole argument.

Fixpoint \( \text{repeat\_free} \, l : \text{Prop} := \)
\( \text{match} \, l \, \text{with} \)
\( \quad \text{nil} \Rightarrow \text{True} \)
\( \quad x :: l' \Rightarrow \neg \text{In} \, x \, l' \land \text{repeat\_free} \, l' \)
\( \text{end}. \)

If equality is middle-excluding on \( A \) then a \( \text{repeat\_free} \) list included in another list is not longer than the other list. This is proved by induction on the \( \text{repeat\_free} \) list. For the inductive step, invoke \( \text{Inelim\_right} \) to decompose the other list along the head of the \( \text{repeat\_free} \) list.

Lemma \( \text{repeat\_free\_incl\_length} : \text{eq\_midex} \rightarrow \forall \, l \, l', \)
\( \text{repeat\_free} \, l \rightarrow \text{incl} \, l \, l' \rightarrow \text{length} \, l \leq \text{length} \, l'. \)

4 On Relations

4.1 Transitive Closure in the Coq Standard Library

Traditionally, the transitive closure of a binary relation is the smallest transitive binary relation including the original relation. The notion of transitive closure can be formally defined by induction, like in the Coq Standard Library. The following function \( \text{clos\_trans} \) waits for a relation over \( A \) and yields its transitive closure, which is also a relation over \( A. \)

Inductive \( \text{clos\_trans} \, R : A \rightarrow A \rightarrow \text{Prop} := \)
\( \quad \text{t\_step} : \forall \, x \, y, \, R \, x \, y \rightarrow \text{clos\_trans} \, R \, x \, y \)
\( \quad \text{t\_trans} : \)
\( \forall \, x \, y \, z, \, \text{clos\_trans} \, R \, x \, y \rightarrow \text{clos\_trans} \, R \, y \, z \rightarrow \text{clos\_trans} \, R \, x \, z. \)

Intuitively, two elements are related by the transitive closure of a binary relation if one can start at the first element and reach the second one in finitely many steps of the original relation. Therefore replacing \( \text{clos\_trans} \, R \, x \, y \rightarrow \text{clos\_trans} \, R \, y \, z \rightarrow \text{clos\_trans} \, R \, x \, z \) by \( R \, x \, y \rightarrow \text{clos\_trans} \, R \, y \, z \rightarrow \text{clos\_trans} \, R \, x \, z \) or \( \text{clos\_trans} \, R \, x \, y \rightarrow R \, y \, z \rightarrow \text{clos\_trans} \, R \, x \, z \) in the definition of \( \text{clos\_trans} \) would yield two relations coinciding with \( \text{clos\_trans}. \) Those three relations are yet different in intension: only \( \text{clos\_trans} \) captures the meaning of the terminology “transitive closure”.

In addition, this paper needs the notion of subrelation.

Definition \( \text{sub\_rel} \, R \, R' : \text{Prop} := \forall \, x \, y, \, R \, x \, y \rightarrow R' \, x \, y. \)
The next lemma asserts that a transitive relation contains its own transitive closure (they actually coincide).

**Lemma** transitive sub rel clos trans : \( \forall R, \) transitive \( R \rightarrow \) sub rel (clos trans \( R \)).

**Proof** Let \( R \) be a transitive relation over \( A \). Prove the subrelation property by the induction principle of clos trans. The base case is trivial and the inductive case follows from the transitivity of \( R \).

\( \square \)

### 4.2 Irreflexivity

A relation is irreflexive if no element is related to itself. Therefore irreflexivity of a relation implies irreflexivity of any subrelation.

**Definition** irreflexive \( R : \) Prop := \( \forall x, \neg R x x \).

**Lemma** irreflexive preserved : \( \forall R R', \) sub rel \( R R' \rightarrow \) irreflexive \( R' \rightarrow \) irreflexive \( R \).

### 4.3 Restrictions

Throughout this paper, finite “subsets” of \( A \) are represented by lists over \( A \). For that specific use of lists, the number and the order of occurrences of elements in a list are irrelevant. Let \( R \) be a binary relation over \( A \) and \( l \) be a list over \( A \). The binary relation restriction \( R l \) relates elements that are both occurring in \( l \) and related by \( R \). The predicate is restricted says that “the support of the given binary relation \( R \) is included in the list \( l \)”. And the next lemma shows that transitive closure preserves restriction to a given finite set.

**Definition** restriction \( R l x y : \) Prop := In x l \& In y l \& R x y.

**Definition** is restricted \( R l : \) Prop := \( \forall x y, R x y \rightarrow \) In x l \& In y l.

**Lemma** restricted clos trans : \( \forall R l, \) is restricted \( R l \rightarrow \) is restricted (clos trans \( R \)) \( l \).

**Proof** Assume that \( R \) is restricted to \( l \). Let \( x \) and \( y \) in \( A \) be such that clos trans \( R x y \), and prove by induction on that last hypothesis that \( x \) and \( y \) are in \( l \). The base case, where “clos trans \( R x y \) comes from \( R x y \)”, follows by definition of restriction. For the inductive case, where “clos trans \( R x y \) comes from clos trans \( R x z \) and clos trans \( R z y \) for some \( z \) in \( A \)”, induction hypotheses are In x l \& In z l and In z l \& In y l, which allows to conclude. \( \square \)

If the support of a relation involves only two (possibly equal) elements, and if those two elements are related by the transitive closure, then they are also related by the original relation. By the induction principle for clos trans and lemma restricted clos trans.

**Lemma** clos trans restricted pair : \( \forall R x y, \) is restricted \( R (x::y::nil) \rightarrow \) clos trans \( R x y \rightarrow R x y. \)
5 On Paths and Transitive Closure

5.1 Paths

The notion of path relates to one interpretation of transitive closure. Informally, a path is a list recording consecutive steps of a given relation. The following predicate says that a given list is a path between two given elements with respect to a given relation.

\[
\text{is\_path } R x y l \{ \text{struct } l \} : \text{Prop} := \\
\begin{align*}
\text{match } l & \text{ with} \\
\text{— nil } \Rightarrow & \ R x y \\
\text{— } z :: l' & \Rightarrow & \ R x z \land \text{is\_path } R z y l' \\
\end{align*}
\]

The following two lemmas show the correspondence between paths and transitive closure. The first is proved by the induction principle of clos\_trans and an appending property on paths proved by induction on lists. For the second, let \( y \) be in \( A \) and prove \( \forall l x, \text{is\_path } R x y l \rightarrow \text{clos\_trans } R x y \) by induction on \( l \).

**Lemma clos\_trans\_path**: \( \forall x y, \text{clos\_trans } R x y \rightarrow \exists l, \text{is\_path } R x y l. \)

**Lemma path\_clos\_trans**: \( \forall y l x, \text{is\_path } R x y l \rightarrow \text{clos\_trans } R x y. \)

Assume that equality is middle-excluding on \( A \) and consider a path between two points. Between those two points there is a repeat\_free path avoiding them and (point-wise) included in the first path. The inclusion is also arc-wise by construction, but this paper does not need it.

**Lemma path\_repeat\_free\_length** : \( \forall x y l, \text{is\_path } R x y l \rightarrow \exists l', -\text{In } x l' \land -\text{In } y l' \land \text{repeat\_free } l' \land \text{length } l' \leq \text{length } l \land \text{incl } l' l \land \text{is\_path } R x y l'. \)

**Proof** Assume that equality is middle-excluding on \( A \), let \( y \) be in \( A \), and perform an induction on \( l \). For the inductive step, call \( a \) the head of \( l \). If \( a \) equals \( y \) then the empty list is a witness for the existential quantifier. Now assume that \( a \) and \( y \) are distinct. Use the induction hypothesis with \( a \) and get a list \( l' \). Case split on \( x \) occurring in \( l' \). If \( x \) occurs in \( l' \) then invoke lemma \( \text{In\_elim\_right} \) and decompose \( l' \) along \( x \), and get two lists. In order to prove that the second list, where \( x \) does not occur, is a witness for the existential quantifier, notice that splitting a path yields two paths (\( a \) priori between different elements) and that appending reflects the repeat\_free predicate (if the appending of two lists is repeat\_free then the original lists also are). Next, assume that \( x \) does not occur in \( l' \). If \( x \) equals \( a \) then \( l' \) is a witness for the existential quantifier. If \( x \) and \( a \) are distinct then \( a::l' \) is a witness. \( \square \)
5.2 Bounded Paths

Given a relation and a natural number, the function $\text{bounded\_path}$ returns a relation saying that there exists a path of length at most the given natural number between two given elements.

Inductive $\text{bounded\_path} \, R \, n : A \rightarrow A \rightarrow \text{Prop}$ :=
$- \text{bp\_intra} : \forall \, x \, y \, l, \text{length} \, l \leq n \rightarrow \text{is\_path} \, R \, x \, y \, l \rightarrow \text{bounded\_path} \, R \, n \, x \, y.$

Below, two lemmas relate $\text{bounded\_path}$ and $\text{clos\_trans}$. The first one follows from $\text{path\_clos\_trans}$; the second one from $\text{clos\_trans\_path}$, $\text{path\_repeat\_free\_length}$, $\text{repeat\_free\_incl\_length}$, and a path of a restricted relation being included in the support of the relation. Especially, the second lemma says that in order to know whether two elements are related by the transitive closure of a restricted relation, it suffices to check whether there is, between those two elements, a path of length at most the “cardinal” of the support of the relation.

Lemma $\text{bounded\_path\_clos\_trans} : \forall \, R \, n, \text{sub\_rel} \, (\text{bounded\_path} \, R \, n) \, (\text{clos\_trans} \, R)$.

Lemma $\text{clos\_trans\_bounded\_path} : \text{eq\_midex} \rightarrow \forall \, R \, l, \text{is\_restricted} \, R \, l \rightarrow \text{sub\_rel} \, (\text{clos\_trans} \, R) \, (\text{bounded\_path} \, R \, (\text{length} \, l))$.

5.3 Restriction, Decidability, and Transitive Closure

The following lemma says that it is decidable whether one step of a given decidable relation from a given starting point to some point $z$ in a given finite set and one step of another given decidable relation from the same point $z$ can lead to another given ending point. Moreover such an intermediate point $z$ is computable when it exists, hence the syntax $\{z : A \mid \ldots\}$.

Lemma $\text{dec\_lem} : \forall \, R' \, R'' \, x \, y \, l, \text{rel\_dec} \, R' \rightarrow \text{rel\_dec} \, R'' \rightarrow \{z : A \mid \text{In} \, z \, l \land R' \, x \, z \land R'' \, z \, y\} + \{z : A, \text{In} \, z \, l \land R' \, x \, z \land R'' \, z \, y\}$.

The following lemma is the middle-excluding version of the previous lemma.

Lemma $\text{midex\_lem} : \forall \, R' \, R'' \, x \, y \, l, \text{rel\_midex} \, R' \rightarrow \text{rel\_midex} \, R'' \rightarrow (\exists \, z : A, \text{In} \, z \, l \land R' \, x \, z \land R'' \, z \, y) \lor (\exists \, z : A, \text{In} \, z \, l \land R' \, x \, z \land R'' \, z \, y)$.

Proof By induction on $l$. For the inductive step, call $a$ the head of $l$. Then case split on the induction hypothesis. In the case of existence, any witness for the induction hypothesis is also a witness for the wanted property. In the case of non-existence, case split on $R' \, x \, a$ and $R'' \, a \, y$.

By unfolding the definition $\text{rel\_midex}$, the next result implies that given a restricted and middle-excluding relation, a given natural number and two given points, either there is a path of length at most that number between those points or there is no such path. Replacing $\text{midex}$ by $\text{dec}$ in the lemma yields a correct lemma about decidability.

Lemma $\text{bounded\_path\_midex} : \forall \, R \, l \, n,$
is\_restricted \, R \, l \rightarrow \text{rel\_midex} \, R \rightarrow \text{rel\_midex} \, (\text{bounded\_path} \, R \, n).

**Proof**  First prove three simple lemmas relating \text{bounded\_path}, \, n, and \, S \, n. Then let \, R \, be a middle-excluding relation restricted to \, l \, and \, x \, and \, y \, be in \, A. Perform an induction on \, n. For the inductive step, case split on the induction hypothesis with \, x \, and \, y. If \text{bounded\_path} \, R \, n \, x \, y \, holds then it is straightforward. If its negation holds then case split on \emph{em\_lem} with \, R, \, \text{bounded\_path} \, R \, n, \, x, \, y, \, and \, l. In the existence case, just notice that a path of length less than \, n \, is of length less than \, S \, n. In the non-existence case, show the negation of \text{bounded\_path} in the wanted property. \hfill \square

Let equality and a restricted relation be middle-excluding over \, A, then the transitive closure of the relation is also middle-excluding. The proof invokes \text{bounded\_path\_midex}, \, \text{bounded\_path\_clos\_trans}, \, and \, \text{clos\_trans\_bounded\_path}. The decidability version of it is also correct.

**Lemma** \text{restricted\_midex\_clos\_trans\_midex} : \text{eq\_midex} \rightarrow \forall \, R \, l, \text{rel\_midex} \, R \rightarrow \text{is\_restricted} \, R \, l \rightarrow \text{rel\_midex} \, (\text{clos\_trans} \, R).

### 5.4 Intermediate Results

The following theorems state the equivalence between decidability of a relation and decidability of the transitive closures of its finite restrictions. The first result invokes \text{clos\_trans\_restricted\_pair} and the second implication uses \text{restricted\_dec\_clos\_trans\_dec}. Note that decidable equality is required only for the second implication. These results remain correct when considering excluded middle instead of decidability.

**Theorem** \text{clos\_trans\_restriction\_dec\_R\_dec} : \forall \, R \, (\forall \, l, \, \text{rel\_dec} \, (\text{clos\_trans} \, (\text{restriction} \, R \, l))) \rightarrow \text{rel\_dec} \, R.

**Theorem** \text{R\_dec\_clos\_trans\_restriction\_dec} : \text{eq\_dec} \rightarrow \forall \, R \, \text{rel\_dec} \, R \rightarrow \forall \, l, \, \text{rel\_dec} \, (\text{clos\_trans} \, (\text{restriction} \, R \, l)).

### 6 Linear Extension and Topological Sorting

Consider \, R \, a binary relation over \, A \, and \, l \, a list over \, A. This section presents a way of preserving acyclicity of \, R \, while “adding arcs” to the restriction of \, R \, to \, l \, in order to build a total and transitive relation over \, l. In particular, if \, R \, is acyclic, then its image by the relation completion procedure must be a strict total order. The basic idea is to compute the transitive closure of the restriction of \, R \, to \, l, \, add an arc if it can be done without creating any cycle, take the transitive closure, add an arc if possible, etc. All those steps preserve existing arcs, and since \, l \, is finite there are finitely many eligible arcs, therefore the process terminates. This is not the fastest topological sort algorithm but its fairly simple expression leads to a simple proof of correctness.
6.1 Total

$R$ is said to be total on $l$ if any two distinct elements in $l$ are related either way. Such a trichotomy property for a relation implies trichotomy for any bigger relation.

Definition trichotomy $R x y : \text{Prop} := R x y \lor x=y \lor R y x$.

Definition total $R l : \text{Prop} := \forall x y, \text{In} x l \rightarrow \text{In} y l \rightarrow \text{trichotomy} R x y$.

Lemma trichotomy_preserved : $\forall R R' x y$, $\text{sub_rel} R R' \rightarrow \text{trichotomy} R x y \rightarrow \text{trichotomy} R' x y$.

6.2 Try Add Arc

If $x$ and $y$ are equal or related either way then define the relation $\text{try_add_arc}$ $R x y$ as $R$, else define it as the disjoint union of $R$ and the arc $(x,y)$.

Inductive $\text{try_add_arc} R x y : A \rightarrow A \rightarrow \text{Prop} :=$

- keep : $\forall z t, R z t \rightarrow \text{try_add_arc} x y z t$
- $\text{try_add} : \neg x=y \rightarrow \neg R y x \rightarrow \text{try_add_arc} x y x y$.

Prove by induction on $l$ and a few case splittings that, under some conditions, a path with respect to an image of $\text{try_add_arc}$ is also a path with respect to the original relation.

Lemma path_\text{try_add_arc_path} : $\forall R t x y l z$, $\neg(x=z \lor \text{In} x l) \lor \neg(y=t \lor \text{In} y l) \rightarrow$

$\text{is_path} R (\text{try_add_arc} R x y) z t l \rightarrow \text{is_path} R z t l$.

The next three lemmas lead to the conclusion that the function $\text{try_add_arc}$ does not create cycles. The first one follows from a few case splittings and the last one highly relies on the second one but also invokes $\text{clos_trans_path}$.

Lemma trans_\text{try_add_arc_sym} : $\forall R x y z t$, $\text{transitive} R \rightarrow \text{try_add_arc} x y z t \rightarrow \text{try_add_arc} x y t z \rightarrow R z z$.

Lemma trans_bounded_path_\text{try_add_arc} : $\text{eq_midx} x y z n$, $\text{transitive} R \rightarrow \text{bounded_path} (\text{try_add_arc} x y) n z z \rightarrow R z z$.

Proof By induction on $n$. The base case requires only $\text{trans_\text{try_add_arc_sym}}$. For the inductive case, consider a path of length less than or equal to $n+1$ and build one of length less than $n+1$ as follows. By $\text{path_repeat_free_length}$ the path may be $\text{repeat_free}$, i.e., without circuit. Proceed by case splitting on the construction of the path: when the path is $\text{nil}$, it is straightforward. If the length of the path is one then invoke $\text{sub_rel_\text{try_add_arc} trans_\text{try_add_arc_sym}}$ else perform a 4-case splitting (induced by the disjunctive definition of $\text{try_add_arc}$) on the first two ($\text{try_add_arc} x y$)-steps of the path. Two cases out of the four need lemmas $\text{transitive_\text{sub_rel_clos_trans}}, \text{path_clos_trans}$, and $\text{path_\text{try_add_arc_path}}$. \hfill $\square$
Lemma try_add_arc_irrefl : eq_midex → ∀ R x y, transitive R → irreflexive R → irreflexive (clos_trans (try_add_arc x y)).

6.3 Try Add Arc (One to Many)

The function try_add_arc_one_to_many recursively tries to (by preserving acyclicity) add all arcs starting at a given point and ending in a given list.

Fixpoint try_add_arc_one_to_many R x l {struct l} : A → A → Prop :=
match l with
  | nil ⇒ R
  | y: l’ ⇒ clos_trans (try_add_arc (try_add_arc_one_to_many R x l’) x y)
end.

The following three lemmas prove preservation properties about the function try_add_arc_one_to_many: namely, arc preservation, restriction preservation, and middle-exclusion preservation. Decidability preservation is also correct, although not formally stated here.

Lemma sub_rel_try_add_arc_one_to_many : ∀ R x l, sub_rel R (try_add_arc_one_to_many R x l).

Proof By induction on l. For the inductive step, call a the head of l and l’ its tail. Use transitivity of sub_rel with try_add_arc_one_to_many x l’ and try_add_arc (try_add_arc_one_to_many R x l’) x a. Also invoke clos_trans and a similar arc preservation property for try_add_arc.

Lemma restricted_try_add_arc_one_to_many : ∀ R l x l’, In x l → incl l’ l → is_restricted R l → is_restricted (try_add_arc_one_to_many R x l’) l.

Proof By induction on l’. Also invoke restricted_try_add_arc_one_to_many, restricted_midex_clos_trans midex with l, and a similar middle-exclusion preservation property for try_add_arc.

Next, a step towards totality.

Lemma try_add_arc_one_to_many_trichotomy : eq_midex → ∀ R x y l l’, In y l’ → In x l → incl l’ l → is_restricted R l → rel_midex R → trichotomy (try_add_arc_one_to_many R x l’) x y.

Proof By induction on l’. For the inductive step, invoke trichotomy_preserved, case split on y being the head of l’ or y occurring in the tail of l’. Also refer to a similar trichotomy property for try_add_arc.
6.4 Try Add Arc (Many to Many)

The function \texttt{try\_add\_arc\_many\_to\_many} requires a relation and two lists. Then, using \texttt{try\_add\_arc\_one\_to\_many}, it recursively tries to safely add all arcs starting in first list argument and ending in the second one.

Fixpoint \texttt{try\_add\_arc\_many\_to\_many} \(R l l\) \{ struct \(l\) \}: \(A \rightarrow A \rightarrow \text{Prop} :=\)
\begin{align*}
&\text{match } l \text{ with} \\
&\quad \text{nil } \Rightarrow R \\
&\quad x:\!\!l' \Rightarrow \text{try\_add\_arc\_one\_to\_many} (\text{try\_add\_arc\_many\_to\_many} R l l') x l \\
&\text{end.}
\end{align*}

The following three results proved by induction on the list \(l\)'s state arc, restriction, and decidability preservation properties of \texttt{try\_add\_arc\_many\_to\_many}.

For the inductive case of the first lemma, call \(l\) the tail of \(l\)', apply the transitivity of \texttt{sub\_rel} with \(\text{try\_add\_arc\_many\_to\_many} R l l\)', and invoke lemma \(\text{sub\_rel}\text{try\_add\_arc\_one\_to\_many}\). Use \texttt{restricted\_try\_add\_arc\_one\_to\_many} for the second lemma. For the third one invoke \(\text{try\_add\_arc\_one\_to\_many\_dec}\) and \(\text{restricted\_try\_add\_arc\_many\_to\_many}\). Middle-exclusion preservation is also correct, although not formally stated here.

\begin{enumerate}
\item Lemma \texttt{sub\_rel\_try\_add\_arc\_many\_to\_many} \(\forall R l l',\text{sub\_rel} R (\text{try\_add\_arc\_many\_to\_many} R l l')\).
\item Lemma \texttt{restricted\_try\_add\_arc\_many\_to\_many} \(\forall R l l',\text{incl} l l' l \rightarrow \text{is\_restricted} R l \rightarrow \text{is\_restricted} (\text{try\_add\_arc\_many\_to\_many} R l l') l\).
\item Lemma \texttt{try\_add\_arc\_many\_to\_many\_dec} \(\forall R l l',\text{incl} l l' l \rightarrow \text{is\_restricted} R l \rightarrow \text{rel\_dec} R \rightarrow \text{rel\_dec} (\text{try\_add\_arc\_many\_to\_many} R l l')\).
\end{enumerate}

The next two results state a trichotomy property and that the function \texttt{try\_add\_arc\_many\_to\_many} does not create any cycle.

\begin{enumerate}
\item Lemma \texttt{try\_add\_arc\_many\_to\_many\_trichotomy} \(\text{eq\_midex} \rightarrow \forall R l x y l',\text{incl} l l' l \rightarrow \text{In} y l \rightarrow \text{In} x l' \rightarrow \text{restricted} R l \rightarrow \text{rel\_midex} R \rightarrow \text{trichotomy} (\text{try\_add\_arc\_many\_to\_many} R l l') x y\).
\end{enumerate}

**Proof** By induction on \(l\). Start the inductive step by case splitting on \(x\) being the head of \(l\)' or occurring in its tail \(l\)'. Conclude the first case by \texttt{try\_add\_arc\_one\_to\_many\_trichotomy}, \texttt{try\_add\_arc\_many\_to\_many\_midex}, and \texttt{restricted\_try\_add\_arc\_many\_to\_many}. Use \texttt{trichotomy\_preserved}, the induction hypothesis, and \texttt{sub\_rel\_try\_add\_arc\_one\_to\_many} for the second case.

\begin{enumerate}
\item Lemma \texttt{try\_add\_arc\_many\_to\_many\_irrefl} \(\text{eq\_midex} \rightarrow \forall R l l',\text{incl} l l' l \rightarrow \text{is\_restricted} R l \rightarrow \text{transitive} A R \rightarrow \text{irreflexive} R \rightarrow \text{irreflexive} (\text{try\_add\_arc\_many\_to\_many} R l l')\).
\end{enumerate}

**Proof** By induction on \(l\). For the inductive step, first prove a similar irreflexivity property for \texttt{try\_add\_arc\_one\_to\_many} by induction on lists and \texttt{try\_add\_arc\_irrefl}. Then invoke \texttt{restricted\_try\_add\_arc\_many\_to\_many}. Both this proof and the one for \texttt{try\_add\_arc\_one\_to\_many} also require transitivity of the transitive closure and an additional case splitting on \(l\)' being \(\text{nil}\) or not.
6.5 Linear Extension/Topological Sort Function

Consider the restriction of a given relation to a given list. The following function tries to add all arcs both starting and ending in that list to that restriction while preserving acyclicity.

**Definition** $LETS \; R \; l : A \rightarrow A \rightarrow Prop := try_{add\_are\_many\_to\_many} (clos\_trans (restriction \; R \; l)) \; l \; l$.

The next three lemmas are proved by $sub\_rel\_try_{add\_are\_many\_to\_many}$, $transitive\_clos\_trans$, and $restricted\_try_{add\_are\_many\_to\_many}$ respectively.

**Lemma** $LETS_{sub\_rel} : \forall \; R \; l, sub\_rel (clos\_trans (restriction \; R \; l)) \; (LETS \; R \; l)$.

**Lemma** $LETS_{transitive} : \forall \; R \; l, transitive (LETS \; R \; l)$.

**Lemma** $LETS_{restricted} : \forall \; R \; l, is_{restricted} (LETS \; R \; l)$.

Under middle-excluding equality, the finite restriction of $R$ to $l$ has no cycle iff $LETS \; R \; l$ is irreflexive. Prove left to right by $try_{add\_are\_many\_to\_many\_irrefl}$, and right to left by $irreflexive\_preserved$ and $LETS_{sub\_rel}$.

**Lemma** $LETS_{irrefl} : eq_{midex} \rightarrow \forall \; R \; l, (irreflexive (clos\_trans (restriction \; R \; l))) \leftrightarrow irreflexive (LETS \; R \; l)$.

If $R$ and equality on $A$ are middle-excluding then $LETS \; R \; l$ is total on $l$. By $R_{\mid midex} \; clos\_trans\_restriction\_midex$ (from the first main result) and $try_{add\_are\_many\_to\_many\_trichotomy}$.

**Lemma** $LETS_{total} : eq_{midex} \rightarrow \forall \; R \; l, rel_{midex} \; R \rightarrow total (LETS \; R \; l)$.

The next two lemmas show that if $R$ and equality on $A$ are middle-excluding (resp. decidable) then so is $LETS \; R \; l$: by $try_{add\_are\_many\_to\_many\_midex}$ (resp. $try_{add\_are\_many\_to\_many\_dec}$) and $R_{\mid midex} \; clos\_trans\_restriction\_midex$ (resp. $R_{\mid dec} \; clos\_trans\_restriction\_dec$).

**Lemma** $LETS_{midex} : eq_{midex} \rightarrow \forall \; R \; l, rel_{midex} \; R \rightarrow rel_{midex} (LETS \; R \; l)$.

**Lemma** $LETS_{dec} : eq_{dec} \rightarrow \forall \; R, rel_{dec} \; R \rightarrow \forall \; l, rel_{dec} (LETS \; R \; l)$.

6.6 Linear Extension

Traditionally, a linear extension of a partial order is a total order including the partial order. Below, a linear extension (over a list) of a binary relation is a strict total order (over the list) that is bigger than the original relation (restricted to the list).

**Definition** $linear\_extension \; R \; l \; R' := is_{restricted} \; R' \; l \wedge sub\_rel \; (restriction \; R \; l) \; R' \wedge transitive \; A \; R' \wedge irreflexive \; R' \wedge total \; R' \; l$. 

The next two lemmas say that a relation “locally” contained in some acyclic relation is “globally” acyclic and that if for any list over $A$ there is a middle-excluding total order over that list, then equality is middle-excluding on $A$.

**Lemma local_global_acyclic** :
$$\forall R, \exists R', \text{sub_rel}\ (\text{restriction } R \ i l)\ R' \land \text{transitive } R' \land \text{irreflexive } R' \rightarrow \text{irreflexive } (\text{clos_trans } R).$$

**Proof** Let $R$ be a relation over $A$. Assume that any finite restriction of $R$ is included in some strict partial order. Let $x$ be in $A$ such that $\text{clos_trans } R x x$. Then derive $\text{False}$ as follows. Invoke $\text{clos_trans_path}$ and get a path. It is still a path for the restriction of $R$ to the path itself (the path is a list seen as a subset of $A$). Use $\text{path_clos_trans}$, then the main assumption, $\text{transitive_sub_rel_clos_trans}$, and the monotonicity of $\text{clos_trans}$ with respect to $\text{sub_rel}$. □

**Lemma total_order_eq_midex** :
$$\forall l, \exists R, \text{transitive } R \land \text{irreflexive } R \land \text{total } R \ i l \land \text{rel_midex } R \rightarrow \text{eq_midex}.$$

**Proof** Assume the left conjunct, let $x$ and $y$ be in $A$, use the assumption with $x::y::\text{nil}$, get a relation, and double case split on $x$ and $y$ being related either way. □

Consider a middle-excluding relation on $A$. It is acyclic and equality is middle-excluding on $A$ iff for any list over $A$ there exists, on the given list, a decidable strict total order containing the original relation.

**Theorem linearly_extendable** :
$$\forall R, \text{rel_midex } R \rightarrow (\text{eq_midex } \land \text{irreflexive } (\text{clos_trans } R) \iff \forall l, \exists R', \text{linear_extension } R \ i l \land \text{rel_midex } R').$$

**Proof** Left to right: by the relevant lemmas of subsection 6.5, $(\text{LETS } R \ i l)$ is a witness for the existential quantifier. Right to left by $\text{local_global_acyclic}$ and $\text{total_order_eq_midex}$. □

### 6.7 Topological Sorting

In this subsection, excluded-middle results of subsection 6.6 are translated into decidability results and augmented: as there is only one concept of linear extension in subsection 6.6, this section presents three slightly different concepts of topological sort. Instead of the equivalence of theorem $\text{linearly_extendable}$, those three definitions yield a quadruple equivalence.

From now on a decidable relation may be represented by a function to booleans instead of a function to $\text{Prop}$ satisfying the definition $\text{rel_dec}$. However, those two representations are “equivalent” thanks to lemmas $\text{rel_dec_bool}$ and $\text{bool_rel_dec}$ in subsection 2.

In this article, a given relation over $A$ is said to be non-uniformly (topologically) sortable if the restriction of the relation to any list has a decidable linear extension.
Acyclicity and Finite Linear Extendability: a Formal and Constructive Equivalence

Definition \(\text{non\textunderscore uni\textunderscore topo\textunderscore sortable} R :=\)
\[ \forall l, \exists R' : A \rightarrow A \rightarrow \text{bool}, \text{linear\_extension} R \ l (\text{fun} \ x \ y \Rightarrow R' \ x \ y=\text{true}). \]

In the definition above, \(R'\) represents a decidable binary relation that intends to be a linear extension of \(R\) over the list \(l\). But \(R'\) has type \(A \rightarrow A \rightarrow \text{bool}\) so it cannot be used with the predicate \(\text{linear\_extension} R \ l\) that waits for an object of type \(A \rightarrow A \rightarrow \text{Prop}\), which is the usual type for representing binary relations in Coq. The function \(\text{fun} \ x \ y \Rightarrow (R' \ x \ y)=\text{true}\) above is the translation of \(R'\) in the suitable type/representation. It waits for two elements \(x\) and \(y\) in \(A\) and returns the proposition \(R' \ x \ y=\text{true}\), in \(\text{Prop}\).

In this article, a given relation over \(A\) is said to be uniformly sortable if there exists a computable function waiting for a list over \(A\) and producing, over the list argument, a (decidable) linear extension of the original relation.

Definition \(\text{uni\textunderscore topo\textunderscore sortable} R := \{ F : \text{list} \ A \rightarrow A \rightarrow \text{bool} \mid \forall l, \text{linear\_extension} R \ l (\text{fun} \ x \ y \Rightarrow (F \ l \ x \ y)=\text{true}) \} \).

The third definition of topological sort requires the concept of \(\text{asymmetry}\), which is now informally introduced: from an algorithmic viewpoint: given a way of representing binary relations, different objects may represent the same binary relation; from a logical viewpoint: two binary relations different in intension, \(i.e.\) their definitions intend different things, may still coincide, \(i.e.\) may be logically equivalent. In an arbitrary topological sort algorithm, the returned linear extension may depend on which object has been chosen to represent the original binary relation. For example, given a two-element set, a topological sort algorithm that is input the empty relation on the given set may produce the two possible linear extensions depending on the order in which the two elements constituting the set are given. This remark leads to the following definition.

Definition \(\text{asym} R G := \forall x,y : A, x\neq y \Rightarrow \neg R \ x \ y \rightarrow \neg R \ y \ x \rightarrow \neg (G \ (x::\text{nil}) \ x \ y \land G \ (y::\text{nil}) \ x \ y)\).

Next comes the definition of asymmetry for a topological sort of a binary relation. The syntax \(\text{let variable:= formula in formula'}\) avoids writing \(\text{formula}\) several times in \(\text{formula'}\).

Definition \(\text{asym\_topo\_sortable} R := \{ F : \text{list} \ A \rightarrow A \rightarrow \text{bool} \mid \text{let} \ G := (\text{fun} \ l \ x \ y \Rightarrow F \ l \ x \ y=\text{true}) \ \text{in} \ \text{asym} R \ G \land \forall l, \text{linear\_extension} R \ l (G \ l) \}\).

Given a binary relation \(R\) over \(A\), the remainder of this subsection proves that the four following assertions are equivalent:

1. Equality on \(A\) is decidable, and \(R\) is decidable and acyclic.
2. \(R\) is middle-excluding and asymmetrically sortable.
3. \(R\) is decidable and uniformly sortable.
4. Equality on \(A\) is decidable, and \(R\) is decidable and non-uniformly sortable.

The following lemma says that if there exists a computable function waiting for a list over \(A\) and producing a (decidable) strict total order over \(A\), then equality on \(A\) is decidable. The proof is similar to the one for \(\text{total\_order\_eq\_midex}\).
Lemma total\_order\_eq\_dec :
\{\ F : list A -> A -> bool \ -> \ \forall \ l, let G := fun x y \Rightarrow F l x y=\text{true} in \ transitive A \ G \ \land \ \irreflexive A \ G \ \land \ total \ G l \} \rightarrow eq\_dec A.

Next lemma shows that LETS yields asymmetric topological sort.

Lemma LETS\_asym : \forall R, asym R (LETS R).

Proof Assume all possible premises, especially let x and y be in A. As a preliminary: the hypotheses involve one relation image of restriction and four relations images of try\_add\_arc. Prove that all of them are restricted to x::y::nil. Then perform a few cases splittings and apply clos\_trans\_restricted\_pair seven times.

The quadruple equivalence claimed above follows from rel\_dec\_midex and the six theorems below. The proofs are rather similar to the middle-excluding case in subsection 6.6. The first theorem proves 1 \rightarrow 2 by the relevant lemmas of subsection 6.5 and LETS producing a witness for the computational existence. The second (straightforward) and the third show 2 \rightarrow 3. The fourth (straightforward) and the fifth, proved by total\_order\_eq\_dec, yield 3 \rightarrow 4. The last shows 4 \rightarrow 1 by invoking local\_global\_acyclic.

Theorem possible\_asym\_topo\_sorting : \forall R, eq\_dec A \rightarrow rel\_dec R \rightarrow irreflexive (clos\_trans A R) \rightarrow asym\_topo\_sortable R.

Theorem asym\_topo\_sortable\_uni\_topo\_sortable : \forall R, asym\_topo\_sortable R \rightarrow uni\_topo\_sortable R.

Theorem asym\_topo\_sortable\_rel\_dec : \forall R, rel\_midex R \rightarrow asym\_topo\_sortable R \rightarrow rel\_dec R.

Proof First notice that R is acyclic by local\_global\_acyclic and that equality on A is decidable by total\_order\_eq\_dec. Then let x and y by in A. By decidable equality, case split on x and y being equal. If they are equal then they are not related by acyclicity. Now consider that they are distinct. Thanks to the assumption, get TS an asymmetric topological sort of R. Case split on x and y being related by TS (x::y::nil). If they are not then they cannot be related by R by subrelation property. If they are related then case split again on x and y being related by TS (y::x::nil). If they are not then they cannot be related by R by subrelation property. If they are then they also are by R by the asymmetry property.

Theorem uni\_topo\_sortable\_non\_uni\_topo\_sortable : \forall R, uni\_topo\_sortable R \rightarrow non\_uni\_topo\_sortable R.

Theorem rel\_dec\_uni\_topo\_sortable\_eq\_dec : \forall R, rel\_dec R \rightarrow uni\_topo\_sortable R \rightarrow eq\_dec A.

Theorem rel\_dec\_non\_uni\_topo\_sortable\_acyclic : \forall R, rel\_dec R \rightarrow non\_uni\_topo\_sortable R \rightarrow irreflexive (clos\_trans A R).
7 Conclusion

This paper has given a detailed account on a few facts related to linear extensions of acyclic binary relations. The discussion is based on a formal proof developed with the proof assistant Coq. The three main results are stated again below.

First, a binary relation over a set with decidable/middle-excluding equality is decidable/middle-excluding if and only if transitive closures of its finite restrictions are also decidable/middle-excluding. That theorem is involved in the proof of the second and third main results. Second, consider a middle-excluding relation over an arbitrary domain. It is acyclic and equality on its domain is middle-excluding if and only if any of its finite restriction has a middle-excluding linear extension. Third, consider \( R \) a decidable binary relation over \( A \). The following three propositions are equivalent:

- Equality on \( A \) is decidable and \( R \) is acyclic.
- Equality on \( A \) is decidable and \( R \) is non-uniformly sortable.
- \( R \) is uniformly sortable.

8 Acknowledgement

I thank Pierre Lescanne for his careful reading and helpful comments, as well as Guillaume Melquiond and Victor Poupet for discussions.

References

Verification of Machine Code Implementations of Arithmetic Functions for Cryptography

Magnus O. Myreen, Michael J. C. Gordon

Computer Laboratory, University of Cambridge, Cambridge, UK

Abstract. This report presents a methodology and some preliminary results for verification of machine code implementations of cryptographic operations. Modularity and reusability of proofs is emphasised.

1 Introduction

Cryptography algorithms such as RSA, Diffie-Hellman and elliptic curve cryptography require efficient operations over large natural numbers (>500 bits). Implementations of operations over large numbers are supported by processors through special purpose instructions (sometimes even special purpose coprocessors). Parts of cryptographic operations are therefore often written directly in machine code to make use of the special purpose instructions for high performance. The fact that parts of cryptographic systems are implemented directly in machine code easily lead to ad hoc correctness proofs.

In this report we present a methodology by which we hope to provide theorems and a methodology that make verification of different machine code implementations of cryptographic operations manageable and reusable (even across different architectures). We present two case studies; the first one illustrates the methodology and the second one presents development towards an efficient machine code implementation of Montgomery multiplication (multiplication modulo a large prime number). The work presented in this report has been carried out within the HOL4 theorem prover.

Affeldt and Marti’s paper [1] on verification of SmartMIPS implementations of arithmetic functions (including Montgomery multiplication) motivated our work. Affeldt and Marti have verified a few arithmetic functions in a model of the SmartMIPS machine language augmented with loops (the loops are compiled away after the proof). In this work we attempt to improve on their approach by using a Hoare logic developed for reasoning directly at the level of machine code [12,11]. Other Hoare logics for machine code have been proposed [13,2] and machine code has been verified in different forms [9,3,4,6]. To the best of our knowledge Affeldt and Marti [1] is the only significant effort at verifying optimised arithmetic functions for cryptography. Li, Owens and Slind [8] have developed a proof-producing compiler that they intend to use to generate efficient machine code implementations of cryptographic algorithms.
2 Methodology

The methodology we propose strives towards modularity and reusability of proofs by a three phase strategy: In order to verify or construct a correct implementation of a particular algorithm one will first (i) verify a basic functional version of the algorithm, then (ii) combine and unroll parts of the verified program to build a correct but more realistic functional version of the algorithm; and finally (iii) prove that a piece of machine code implements the functional program produced in step (ii).

The benefit of this approach is that step (i) and (ii) are independent of the target machine architecture and can hence be reused. Also in our experience it is rather easy to prove that the functional programs from step (iii) are implemented by the proposed optimised machine code.

The following table gives some indication that our approach may require less effort than the approach proposed by Affeldt and Marti. The table below gives the number of lines of proof script required for the verification of functional and machine code implementations of the algorithms listed in the left column.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>TFL</th>
<th>ARM</th>
<th>total</th>
<th>A&amp;M</th>
</tr>
</thead>
<tbody>
<tr>
<td>(script at top of file)</td>
<td>120</td>
<td>95</td>
<td>115</td>
<td>—</td>
</tr>
<tr>
<td>addition</td>
<td>135</td>
<td>88</td>
<td>223</td>
<td>835</td>
</tr>
<tr>
<td>subtraction</td>
<td>140</td>
<td>88</td>
<td>228</td>
<td>1473</td>
</tr>
<tr>
<td>multiplication</td>
<td>202</td>
<td>fin</td>
<td>fin</td>
<td>1634</td>
</tr>
<tr>
<td>Montgomery multiplication</td>
<td>864</td>
<td>fin</td>
<td>fin</td>
<td>3881</td>
</tr>
</tbody>
</table>

TFL — number of lines of proof script required in step (i) and (ii)
ARM — number of lines of proof script required in step (iii) for ARM machine code
total — sum of TFL and ARM columns
A&M — total number for Affeldt and Marti’s SmartMIPS implementations [1]
fin — numbers will be included in the final version

A comparison between our numbers and those of Affeldt and Marti is only approximate, since Affeldt and Marti work in Coq rather than HOL4.

3 Case studies

The general methodology is presented first on the simple case of addition and then on the less obvious algorithm for Montgomery multiplication.

3.1 Addition

When constructing a functional implementation of a particular algorithm, we start by defining the elementary operations that we know how to implement in machine code (or can expect to find in the instruction set). For addition we require an operation that performs one step of a large addition. Define single_add to take two bit strings of length α and a boolean carry bit as input and have it return the sum of the inputs as well as a boolean carry-out. Read \( \text{dimword } (:\alpha) \) as \( 2^\alpha \), i.e. it is the number of bits strings that have length/type \( \alpha \).
single_add \(x:\alpha\) \(y:\alpha\) \(c = \)
\((x + y + \text{if } c \text{ then } 1w \text{ else } 0w,\)
\(\text{dimword } (\alpha) \leq w2n x + w2n y + \text{if } c \text{ then } 1 \text{ else } 0)\)

The functional program for addition is then a simple loop:

\[
\begin{align*}
\text{mw\_add } & \text{[]} \text{ ys } c = ([],c) \\
\text{mw\_add } & (x::xs) \text{ (y::ys) } c = \\
& \text{let } (z,c1) = \text{single\_add } x y c \text{ in} \\
& \text{let } (zs,c2) = \text{mw\_add } xs \text{ ys } c1 \text{ in} \\
& (z::zs,c2)
\end{align*}
\]

In order to state the specification for the functional implementations, we define \(n2mw\ i\ n\) to be a list (of length \(i\)) of bits strings (of length \(\alpha\)) such that the concatenation of these bit strings represent the \(i \times \alpha\) least-significant bits of the binary representation of the natural number \(n\).

\[
\begin{align*}
n2mw 0 \ n & = [] \\
n2mw \ (i+1) \ n & = n2w \ n :: n2mw \ i \ (n \ \text{DIV} \ \text{dimword } (\alpha))
\end{align*}
\]

The specification for \(\text{mw\_add}\) is the following. Let \(\text{dimwords } i\ (\alpha) = 2^{i \times \alpha}\) and let \(b2n \ c = \text{if } c \text{ then } 1 \text{ else } 0\).

\[
\forall i \ p \ m \ n \ c.
\text{mw\_add } (n2mw \ i \ m) \ (n2mw \ (i + p) \ n) \ c = \\
(n2mw \ i \ (m + n + b2n \ c), \text{dimwords } i\ (\alpha) \leq \\
m \ MOD \ \text{dimwords } i\ (\alpha) + n \ MOD \ \text{dimwords } i\ (\alpha) + b2n \ c)
\]

The functional program \(\text{mw\_add}\) can be implemented in ARM machine code using a single loop that tests for the end of the sequence using the \text{teq} instruction (an instruction that leaves the carry status bit untouched).

\[
\begin{align*}
\text{L: } & \text{ldr } a, [i], #4 \\
& \text{ldr } b, [j], #4 \\
& \text{adcs } a, a, b \\
& \text{str } a, [k], #4 \\
& \text{teq } i, t \\
& \text{bne } L
\end{align*}
\]

One can prove that the above code implements \(\text{mw\_add}\) by induction on the length of the first argument to \(\text{mw\_add}\) together with the rules of the Hoare logic presented in Myreen and Gordon [12]. The specification which states that the ARM code implements \(\text{mw\_add}\) is shown in Appendix A.

The specification is made legible if we introduce the following definition. Let \(\text{bignum } a\# i\ n\) state that register \(a\) holds an aligned address which points at some location where the sequence \(n2mw \ i\ n\) is stored. Let \(\text{bignum'}\ a\# i\ n\) be the same except that the address points at the location immediately following the sequence \(n2mw \ i\ n\).

\[
\begin{align*}
\text{bignum } a\# i\ n & = \exists x. \ R30 \ a \ x \ * \ ms \ x \ (n2mw \ i\ n) \\
\text{bignum'}\ a\# i\ n & = \exists x. \ R30 \ (a + n2w i) \ * \ ms \ x \ (n2mw \ i\ n)
\end{align*}
\]
Using \texttt{bignum} the above ARM code has the following Hoare-triple specification.

\[
\{ \texttt{bignum i p m \ast bignum j p n \ast bignum k p q \ast}
\]
\[
R a_\ast \ast R b_\ast \ast R30 \ t (i x + n2w p) \ast
\]
\[
carry\_\text{status} c \ast \text{cond} (p \neq 0) \}
\]
\[
(... \text{code} ...)
\]
\[
\{ \texttt{bignum' i p m \ast bignum' j p n \ast bignum' k p (m + n + b2n c) \ast}
\]
\[
R a_\ast \ast R b_\ast \ast R30 \ t (i x + n2w p) \ast
\]
\[
carry\_\text{status} (\text{carry32 p m n c}) \} \}
\]

where \text{carry32 p m n c} abbreviates \(2^{32 \times p} \leq m \mod 2^{32 \times p} + n \mod 2^{32 \times p} + b2n c\).

### 3.2 Montgomery multiplication

Montgomery multiplication is an algorithm that is commonly used in implementations that require multiplication modulo a large prime. Given \(n, r\) and \(r'\) such that \(n < r\), \(\gcd (n, r) = 1\) and \((r \times r') \mod n = 1\), Montgomery multiplication \(\text{monprod}\) calculates the product of \(a, b\) and \(r'\) modulo \(n\):

\[
\text{monprod}(a, b, n) = (a \times b \times r') \mod n, \quad \text{for} \ a < n \text{ and } b < n.
\]

Let \(\bar{a}\) denote \((a \times r) \mod n\). Montgomery multiplication calculates the product of values represented as \(\bar{a}\) and \(\bar{b}\).

\[
\text{monprod}(\bar{a}, \bar{b}, n) = (\bar{a} \times \bar{b} \times r') \mod n
\]
\[
= (a \times r \times b \times r \times r') \mod n
\]
\[
= (a \times r \times b \times 1) \mod n
\]
\[
= (a \times b) \mod n
\]

The conversion from \(\bar{a}\) into \(a \mod n\) can be done using \(\text{monprod}\).

\[
\text{monprod}(\bar{a}, 1, n) = (\bar{a} \times 1 \times r') \mod n
\]
\[
= (a \times r \times r') \mod n
\]
\[
= a \mod n
\]

The conversion from \(a\) into \(\bar{a}\) requires an implementation of modulus.

We will refrain from describing the details of Montgomery multiplication here for that is described well elsewhere: Montgomery describes the the basic algorithm elegantly in [10], Dussé \textit{et al.} [5] and Certin Kaya Koc \textit{et al.} [7] describe optimisations. Instead we will just note that the following implementation of \(\text{monprod}\) called \texttt{mw_monprod},

\[
\texttt{mw\_mul} \ [\ ] \ y \ c = (\ [\ ], c)
\]
\[
\texttt{mw\_mul} \ (x::xs) \ y \ c =
\]
\[
\text{let} \ (z,c1) = \text{single\_mul} \ x \ y \ c \ \text{in}
\]
\[
\text{let} \ (zs,c2) = \text{mw\_mul} \ xs \ y \ c1 \ \text{in}
\]
\((z::zs,c2)\)

\[
\begin{align*}
\text{mw_add_mul} & \ x\ ys\ zs \ = \\
& \ \text{FST} \ (\text{mw_add} \ zs \ (\text{FST} \ (\text{mw_mul} \ ys \ x 0w)) \ F)
\end{align*}
\]

\[
\begin{align*}
\text{mw_monmult} & \ []\ ys\ ns\ m\ zs \ = \ zs \\
& \text{mw_monmult} \ (x::xs)\ ys\ ns\ m\ zs \ = \\
& \quad \text{let } u = (x \ast HD\ ys + HD\ zs) \ast m \text{ in} \\
& \quad \text{let } zs = \text{mw_add_mul} \ x\ ys\ (zs++[0w]) \text{ in} \\
& \quad \text{let } zs = \text{mw_add_mul} \ u\ ns\ zs \text{ in} \\
& \quad \text{mw_monmult} \ xs\ ys\ ns\ m \text{ (TL } zs) \\
\text{mw_monprod} & \ xs\ ys\ ns\ m\ zs \ = \\
& \quad \text{let } \text{zs} = \text{mw_monmult} \ xs\ ys\ ns\ m\ zs \text{ in} \\
& \quad \text{let } (\text{zs}',c) = \text{mw_sub} \ zs\ ns\ T \text{ in} \\
& \quad \text{(if } c \text{ then } \text{zs}' \text{ else } \text{zs})
\end{align*}
\]

satisfies the specification given below. Here we have taken \(r = \text{dimwords} \ i\ \alpha\), which is a power of 2 and hence ‘\(n\) is odd’ is sufficient to guarantee \(gcd(n, r) = 1\).

\[
\forall a\ b\ n\ n'\ r' \ i.
\begin{align*}
(n \times n' = \text{dimwords} \ i\ \alpha \times r' - 1) & \land \text{ODD } n \land 0 < n' \land \\
\text{n < dimwords} \ i\ \alpha & \land a \leq n \land b \leq n \Rightarrow \\
(\text{mw_monprod} \ (n2mw \ i\ a)) & (n2mw \ (i+2) \ b) \ (n2mw \ (i+2) \ n) \ (n2w \ n') \\
(n2mw \ (i+1) \ 0) & = \\
n2mw \ (i+1) \ ((a \times b \times r') \ \text{MOD} \ n)
\end{align*}
\]

**Optimisation 1.** The first version of Montgomery multiplication \(\text{mw_monprod}\) was reasonably easy to prove (≈400 lines of proof script), but the implementation is unsatisfactory in many ways. In what follows we will successively improve the functional implementation towards functional implementations that calculate the value in a less wasteful manner.

The first and obvious improvement is to combine the two occurrences of \(\text{mw_add_mul}\) into one function. A function \(\text{mw_add_mul_mul}\) was constructed which calculates the same value as two applications of \(\text{mw_add_mul}\). An unrolling of the new function is now the body for function \(\text{mw_monmult2}\).

\[
\begin{align*}
\text{mw_monmult2} & \ []\ ys\ ns\ m\ zs \ = \ zs \\
\text{mw_monmult2} \ (x::xs)\ (y::ys)\ (n::ns)\ m\ (z::zs) \ = \\
& \quad \text{let } u = (x \ast y + z) \ast m \text{ in} \\
& \quad \text{let } (w1,c1,b1) = \text{double_mul_add} \ y\ x\ u\ 0w\ 0w\ z \text{ in} \\
& \quad \text{let } (w2,c2,b2) = \text{mw_add_mul_mul} \ ys\ zs\ x\ u\ c1\ b1 \text{ in} \\
& \quad \text{let } (w3,c3,b3) = \text{double_mul_add} \ 0w\ 0w\ x\ u\ c2\ b2 \text{ (LAST } zs) \text{ in} \\
& \quad \text{let } (w4,c4,b4) = \text{double_mul_add} \ 0w\ 0w\ x\ u\ c3\ b3\ 0w \text{ in} \\
& \quad \text{let } zs = \text{zs}++[w3;w4] \text{ in} \\
& \quad \text{mw_monmult2} \ xs\ (y::ys)\ (n::ns)\ m\ zs
\end{align*}
\]

The new definition of \(\text{mw_monprod}\) satisfies the same specification as the original except that now both occurrences of +2 are removed.
Optimisation 2. For the second optimisation we note that the result is one word too long, the result is returned as a list of \(i+1\) words while the result fits into \(i\) words. By looking at the implementation one observes that the last element of the list \(zs\) is always handled separately from the rest of \(zs\). This suggests that it may be beneficial to keep the last element separate throughout the computation (in the machine code implementation we would like to keep the last element in a register). The new implementation simply splits the last element off \(zs\):

\[
\text{mw_monmult3} \; \text{[]} \; \text{ys} \; \text{ns} \; \text{m} \; (zs, z') = (zs, z') \\
\text{mw_monmult3} \; (x::xs) \; (y::ys) \; (n::ns) \; m \; (z::zs, z') = \\
\text{let } u = (x * y + z) * m \text{ in} \\
\text{let } (w1,c1,b1) = \text{double_mul_add} \; y \; n \; x \; u \; 0w \; 0w \; z \; \text{in} \\
\text{let } (w2,c2,b2) = \text{mw_add_mul_mult} \; ys \; ns \; zs \; x \; u \; c1 \; b1 \text{ in} \\
\text{let } (w3,c3,b3) = \text{double_mul_add} \; 0w \; 0w \; 0w \; x \; u \; c2 \; b2 \; z' \; \text{in} \\
\text{let } (w4,c4,b4) = \text{double_mul_add} \; 0w \; 0w \; x \; u \; c3 \; b3 \; 0w \; \text{in} \\
\text{let } (zs, z') = (ws ++ [w3], w4) \text{ in} \\
\text{mw_monmult3} \; zs \; (y::ys) \; (n::ns) \; m \; (zs, z')
\]

The specification of \text{mw_monprod3} is the same as the one for \text{mw_monprod2} except that the result is now of length \(i\) and \((n2mw \; (i+1) \; 0)\) is replaced by \((n2mw \; i \; 0, 0w)\).

Optimisation 3. It would be unfortunate if the final implementation requires the array implementing \(zs\) to be initialised to zero, since that is likely to require a separate loop, which writes zero into each location before calling the algorithm. The requirement of a zeroed input can be removed by unrolling the main loop once in order to break out the part of the program which can assume that \(zs\) is zero, and hence can be implemented more efficiently (fewer load instructions). The new implementation unrolls \text{mw_monmult3} once.

\[
\text{mw_moninit} \; ys \; ns \; x \; m = \\
\text{mw_monmult3_step} \; ys \; ns \; (\text{MAP} \; (\lambda x. 0w) \; ys, 0w) \; x \; m
\]

\[
\text{mw_monprod4} \; (x::xs) \; ys \; ns \; m = \\
\text{let } (zs, z) = \text{mw_moninit} \; ys \; ns \; x \; m \text{ in} \\
\text{let } (zs, z) = \text{mw_monmult3} \; xs \; ys \; ns \; m \; (zs, z) \text{ in} \\
\text{let } (zs', c) = \text{mw_sub} \; zs \; ns \; T \text{ in} \\
\text{let } c' = \text{SNDE} \; (\text{single_sub} \; zs \; 0w \; c) \text{ in} \\
\text{if } c' \text{ then } zs' \text{ else } zs
\]

The new implementation satisfies the following specification.
\[ \forall a \ b \ n \ n' \ r' \ i. \\
(n \times n' = \text{dimwords} i \times r' - 1) \land \text{ODD} n \land 0 < n' \land \\
n < \text{dimwords} i \land a \leq n \land b \leq n \Rightarrow \\
(\text{mw} \_\text{monprod4} (n2mw i a) (n2mw i b) (n2mw i n) (n2w n')) = \\
n2mw i ((a \times b \times r') \text{MOD} n) \]

**ARM implementation.** We have not yet had time to verify an ARM implementation of `mw_monprod4`. [A verified implementation will have been constructed by the time the final version of this report is to be submitted.]

### 4 Further Work

Our aim is to develop machine code implementations of all the operations required for an implementation of elliptic curve cryptography. Another goal is to investigate how these ideas can be applied to architectures other than ARM. We believe that the Hoare logic, which was used to reason about ARM machine code, is readily instantiated to other models of instruction set architectures.

### References


A Specification of Addition

\[
\{ \text{R30 i ix * ms ix xs *} \\
\text{R30 j jx * ms jx ys *} \\
\text{R30 k kx * ms kx zs *} \\
\text{Ra * R b * R30 t (ix + wLENGTH xs) *} \\
\text{carry_statuc c * cond (xs \neq [])*} \\
\text{cond (LENGTH xs = LENGTH zs)* cond (LENGTH zs \leq LENGTH ys)} \}
\]

\[
\{ \text{LDR a, [i], #4 ;} \\
\text{LDR b, [j], #4 ;} \\
\text{ADCS a, a, b ;} \\
\text{STR a, [k], #4 ;} \\
\text{TEQ i, t ;} \\
\text{BNE -20} \}
\]

\[
\{ \text{R30 i (ix + wLENGTH xs) * ms ix xs *} \\
\text{R30 j (jx + wLENGTH xs) * ms jx ys *} \\
\text{R30 k (kx + wLENGTH xs) * ms kx (FST (mw_add xs ys c)) *} \\
\text{Ra * R b * R30 t (ix + wLENGTH xs) *} \\
\text{carry_statuc (SND (mw_add xs ys c))} \}
\]
Translating Haskell to Isabelle

Paolo Torrini, Christoph Lueth
Christian Maeder, Till Mossakowski
Informatik, Universitaet Bremen

Abstract. The Hets-Programatica program-development and proof-management system has been used in order to implement partial translations of Haskell to Isabelle higher-order logics — HOL and HOLCF. Both translations rely on shallow embedding of denotations, though with strong restrictions in the case of HOL. Part of the novelty of our approach is in the use of theory morphisms, as implemented in AWE, to cope with the translation of monadic operators.

Automated translations between programming and specification languages, together with the corresponding integration of analysers and theorem-provers, can provide important support to the formal development and verification of programs. Indeed, the translation of a program to a logic in which requirements can be expressed is preliminary step to any proof of correctness. Important aspects of a translation can be the level of adequacy provided by the semantics on which it rests, as well as the possibility of making the relevant theorem proving as easy as possible. In fact, it has long been argued that functional languages, based on notions closer to general, mathematical ones, can make the task of proving assertions about them easier, owing to the relative clarity and simplicity of their semantics [Tho92].

Haskell is a strongly typed, purely functional language with lazy evaluation, polymorphic types extended with type constructor classes, and a syntax for side effects and pseudo-imperative code based on monadic operators [PJ03]. Isabelle is an SML-written, generic theorem-prover that includes the formalisation of several logics [Pau94]. Here we are presenting automated translations of Haskell to Isabelle higher-order logics, implemented as functions of Hets [MML07], an Haskell-based application designed to support heterogeneous specification and formal development of programs. Hets supplies parsing, static analysis and proof management, as well as with interfaces to various language-specific tools. As far as interactive proofs are concerned, it relies on an interface with Isabelle. Hets relies on Programatica [HHJK04] for the parsing and the static analysis of Haskell programs. Programatica is also an Haskell-specific, formal development system with its own proof management, including a specification logic and translations to different proof tools — to Isabelle as well, though following a different approach from ours [HMW05].

Isabelle-HOL (hereafter HOL) is the implementation in Isabelle of classical higher-order logic based on simply typed lambda calculus extended with axiomatic type classes. It provides support for reasoning about programming functions, both in terms of rich libraries and efficient automa-
In this respect, it has essentially superseded FOL (classical first-order logic) as a standard. HOL has an implementation of recursive total functions based on Knaster-Tarski fixed-point theorem. On the other hand, HOLCF [MNvOS99] is HOL conservatively extended with the logic of computable functions — a formalisation of domain theory. In HOL types can be interpreted as sets (class type); functions are total and may not be computable. A non-primitive recursive function may require discharging proof obligations already at the stage of definition — a specific measure has to be given for the function to be proved monotonic. In HOLCF each type can be interpreted as a pointed complete partially ordered set (class pcpo) i.e. a set with a partial order which is closed w.r.t. \( \omega \)-chains and has a bottom. The Isabelle formalisation, based on axiomatic type classes [Wen05], makes it possible to deal with complete partial orders in quite an abstract way. Functions are generally partial and computability can be expressed in terms of continuity. Recursion can be expressed in terms of least fixed-point operator, and so, in contrast with HOL, function definition does not depend on proofs. Nevertheless, proving theorems in HOLCF may turn out to be comparatively hard. After being spared the need to discharge proof obligations at the definition stage, one has to bear with assumptions on function continuity throughout the proofs. A standard strategy is then to define as much as possible in HOL, using HOLCF type constructors to lift types only when this is necessary.

Although translations of functional languages to first-order systems have been given in the past — those to FOL of Miranda [Tho94, Tho89, HT95] and Haskell [Tho92], both based on large-step operational semantics; the translation of Haskell to the Agda implementation of Martin-Löf type theory in [ABB+05] — still, higher-order logic may be quite helpful in order to deal with features such as currying and polymorphism. Moreover, higher-order approaches may rely on denotational semantics — as for examples, [HMW05] translating Haskell to HOLCF; [LP04] translating ML to HOL — allowing for program representation closer to the specification as well as for proofs comparatively more abstract and general. By shallow embedding we mean one that relies heavily on extra-logical features of the target language, particularly with respect to types and recursion. In contrast, by deep embedding we mean one that relies on the object-level definition of all the relevant notions. The latter may be a plus as to semantic clarity and possibly, provided the logic is expressive enough, to generality. Taking advantage of extra-logical, built-in features, on the other hand, may help make theorem proving less specific and tedious.

The translation of Haskell to HOLCF proposed in [HMW05] uses deep embedding to deal with types. Haskell types are translated to terms, relying on a domain-theoretic modelling of the type system at the object level. The advantage of this approach is that it can capture most features, including type constructor classes — and therefore monads — in a general way. The practical drawback is that plenty of the automation built into Isabelle type checking gets lost. In contrast, the translations that we are presenting here are based on a shallow embedding approach, relying as much as possible on similarities between type systems (based
on Hindley-Milner polymorphism), translating Haskell types to HOLCF and HOL types, respectively, and Haskell classes to Isabelle axiomatic classes, in a comparatively straightforward way. In the case of HOLCF, we expect equivalence between Haskell programs and their translation beyond the level of typeable output. This translation covers a significant part of the Prelude syntax, although there are several limitations, particularly related to built-in types, pattern-matching and local definitions. The translation to HOL is more primitive — essentially, it only covers total primitive recursive program functions. Type constructor classes cannot be dealt with in a general way; however, the AWE implementation of theory morphism [BJL06] can be used in order to translate special cases (see Section 11).

The translation to HOLCF keeps into account partiality, i.e. the fact that a function might be undefined for certain values, either because the definition is missing, or because the program does not terminate. It keeps into account laziness, as well, i.e. the fact that by default function values in Haskell are passed by name and evaluated only when needed. This is done by following, in the main lines, the denotational semantics for lazy evaluation given in [Win93], and also by relying on existing formalisations of computational notions — notably streams and maybe — in HOLCF.

The translation to HOL takes into account neither partiality nor laziness — and then requires a rather strong extra-logical assumption, that we restrict to totally defined program functions. A better semantics could be obtained by lifting the type of function values by \texttt{option}, but this has not been pursued here.

1 Translations in Hets

Hets is a tool set integrating several languages, provers and translations between these, see [Mos05] and [Mos06]. The Haskell-to-Isabelle translation also requires GHC, Programatica, Isabelle and AWE. The application is run by a command that takes as arguments a target logic and an Haskell program, given as a GHC source file. The latter gets analysed and translated, the result of a successful run being an Isabelle theory file in the target logic.

The Hets internal representation of Haskell is similar to that of Programatica, whereas the internal representation of Isabelle is based on the ML definition of the Isabelle base logic, extended in order to allow for a simpler representation of HOL and HOLCF. Haskell programs and Isabelle theories are internally represented as Hets theories — each of them formed by a signature and a set of sentences, according to the theoretical framework described in [Mos05]. Each translation, defined as composition of a signature translation with a translation of all sentences, is essentially a morphism from theories in the internal representation of the source language to those in the representation of the target language. Each translation relies on an Isabelle theory, respectively \texttt{HsHOLCF}, extending HOLCF, and \texttt{HsHOL}, extending HOL, containing specific definitions. The latter in particular uses the AWE implementation of theory morphisms.
2 Naming conventions

Names of types as well as of terms are translated by a renaming function that preserves them, up to avoidance of clashes with Isabelle keywords — indicated with a $t$ subscript further on. There are some reserved names, listed here as they appear in some of the examples.

1) Names for type variables, in the translation to HOL: `vX; any string terminating with $XXn$ where $n$ is an integer.

2) Names for term constants, in the translation to HOL: strings obtained by joining together names of defined functions, using $X$ as end sequence and separator.

3) Names for term variables, in both translations: $pXn$, $qXn$, with $n$ integer.

4) Names for type destructors, in the translation to HOLCF: $Cn$, where $C$ is a data constructor name and $n$ is an integer.

3 HOLCF: Types

Haskell type variables are translated to variables of class pcpo. Built-in types such as Boolean and Integer are translated to the lifting of its corresponding HOL type. The HOLCF type constructor $lift$ is used to lift HOL types to flat domains — in contrast with $u$, which is used to lift domains while preserving their structure. In the case of Boolean, we translate to $tr$, defined in HOLCF as $bool lift$. The types of Haskell functions and product are translated, respectively, to HOLCF function spaces and lazy product — i.e. such that $\bot = (\bot \times \bot) \neq (\bot \times \'a) \neq ('a \times \bot)$, consistently with lazy evaluation. Type constructors are translated to corresponding HOLCF ones (notably, parameters precede type constructors in Isabelle syntax). In particular, lists are translated to the domain $seq$ defined in the IOA library of HOLCF. $Maybe$ is translated to HOLCF-defined $maybe$ (the disjoint union of the lifted unit type and the lifted domain parameter). Each type is associated to a sort in Isabelle, defined by the set of the classes of which it is member.

Type translation to HOLCF, apart from mutual dependencies, may be summed up as follows (where $t$ is a renaming function):

\[
\begin{align*}
[a] &= 'a :: pcpo \\
[Bool] &= tr \\
[Integer] &= int lift \\
[a \to b] &= [a] \to [b] \\
[(a,b)] &= [a] \times [b] \\
[[a]] &= [a] \_seq \\
[Maybe a] &= [a] maybe \\
[TyCons a_1 \ldots a_n] &= [a_1] \ldots [a_n] TyCons_t
\end{align*}
\]

4 HOL: Types

Haskell types are mapped to corresponding HOL ones — thus so for Boolean and Integer. All variables are of class type. HOL function type,
product and list are used to translate the corresponding Haskell constructors. Type translation to HOL, apart from mutual dependencies, may be summed up as follows.

\[
\begin{align*}
\lceil a \rceil &= \prime a :: \text{type} \\
\lceil \text{Bool} \rceil &= \text{bool} \\
\lceil \text{Integer} \rceil &= \text{int} \\
\lceil a \rightarrow b \rceil &= \lceil a \rceil \Rightarrow \lceil b \rceil \\
\lceil (a, b) \rceil &= \lceil a \rceil \times \lceil b \rceil \\
\lceil [a] \rceil &= \lceil a \rceil \text{list} \\
\lceil \text{TyCons} a_1 \ldots a_n \rceil &= \lceil a_1 \rceil \ldots \lceil a_n \rceil \text{TyCons}_t
\end{align*}
\]

5 Signature

Function declarations are associated to the corresponding ones in the target logic, defined by the Isabelle keyword \texttt{consts}. Datatype declarations are associated to the corresponding ones, as well, but here we have some difference. In HOLCF datatype declarations define types of class \texttt{pcpo} by the keyword \texttt{domain} (hence also \texttt{domain declarations}). In HOL they define types of class \texttt{type} by the keyword \texttt{datatype}. Notably, in contrast with Haskell and HOL, HOLCF datatype declarations require an explicit introduction of destructors; these are provided automatically according to the naming pattern in Section 2, point 4. Apart from this aspect, the meta-level features of the the two type translations are essentially similar.

In contrast with Haskell, the order of declarations matters in Isabelle, and this aspect is taken care of automatically in both translations. Translation of mutually recursive datatypes, as the one shown in the following example, relies on specific Isabelle syntax (using the keyword \texttt{and}).

\begin{verbatim}
data AC a b = Ct a | Rt (AC a b) (BC a b) data BC a b = Dt b | St (BC a b) (AC a b)
\end{verbatim}

Translating to HOLCF gives the following.

\begin{verbatim}
domain \langle \prime a :: \text{pcpo}, \prime b :: \text{pcpo} \rangle BC = Dt (\langle \prime b :: \text{pcpo} \rangle BC) \\
\quad St (\langle \prime a :: \text{pcpo}, \prime b :: \text{pcpo} \rangle BC) \\
\text{and } \langle \prime a :: \text{pcpo}, \prime b :: \text{pcpo} \rangle AC = Ct (\langle \prime a :: \text{pcpo} \rangle AC) \\
\quad Rt (\langle \prime a :: \text{pcpo} \rangle AC)
\end{verbatim}

Translating to HOL gives the following.

\begin{verbatim}
datatype \langle \prime a, \prime b \rangle BC = Dt \prime b | St (\langle \prime a, \prime b \rangle BC) \\
\text{and } \langle \prime a, \prime b \rangle AC = Ct \prime a | Rt (\langle \prime a, \prime b \rangle AC)
\end{verbatim}

6 HOLCF: Sentences

Essentially, each function definition is translated to a corresponding one. Non-recursive definitions are translated to standard Isabelle definitions
(introduced by the keyword `defs`), whereas the translation of recursive definitions relies on the HOLCF package `fixrec`. Lambda abstraction is translated as continuous abstraction (`LAM`), function application as continuous application (the `dot` operator). These notions coincide with corresponding ones in HOL, i.e. with lambda abstraction (`λ`) and standard function application, whenever all arguments are continuous.

Terms of built-in type (Boolean and Integer) are translated to lifted HOL values, using the HOLCF-defined lifting function `Def`. The bottom element `⊥` is used for the undefined terms. We use HOLCF-defined lifting function such as `flift1 : : (′a ⇒ ′b :: pcpo) ⇒ (′a lift → ′b lift)` and `flift2 : : (′a ⇒ ′b) ⇒ (′a lift → ′b lift)` in order to lift functions and their arguments. In particular, we use the following operator, defined in HsHOLCF, to map binary arithmetic functions to lifted functions over lifted integers.

```
fliftbin :: (′a ⇒ ′b ⇒ ′c) ⇒ (′al i f t → ′bl i f t → ′cl i f t)
fliftbin f == flift1 (λx. flift2 (fx))
```

Boolean values are translated to values of `tr` — i.e. `TT`, `FF` and `⊥`. Boolean connectives are translated to the corresponding HOLCF lifted operators. HOLCF-defined `If then else fi` and `case` syntax are used to translate conditional and case expressions, respectively. There are some restrictions, however, on the latter, due to limitations in the translation of patterns (see Section 8): in particular, the case term should always be a variable, and no nested patterns are allowed.

The translation of lists relies on the `seq` domain, defined as follows in IOA.

```
domain 'a seq = nil | # (HD :: 'a) (lazy TL :: 'a seq)
```

The keyword `lazy` ensures that `x # # ⊥` is `⊥`, allowing for partial sequences as well as for infinite ones [MNvOS99].

Haskell allows for local definitions by means of `let` and `where` expressions. Those `let` expressions in which the left-hand side is a variable are translated to similar Isabelle ones; neither other `let` expressions (i.e. those containing patterns on the left hand-side) nor `where` expressions are covered. The translation of terms (minus mutual recursion) may be summed up as follows:

- `[x :: a] = x :: [a]`
- `[c] = c`
- `[^x → f] = LAM x. [f]`
- `[a, b] = ([a], [b])`
- `[f a1 ... an] = FIX f1. f1 · [a] ... · [an]` where `f :: τ, f1 :: [τ]`
- `[let x1 ... xn in exp] = let [x1] ... [xn] in [exp]`

In HOLCF all recursive functions can be defined by the fixpoint operator. Coding this directly turns out to be rather cumbersome, particularly in the case of mutually recursive functions, where tuples of defining terms and tupled abstraction would be needed. In contrast,
the \texttt{fixrec} package allows us to handle fixpoint definitions in a way much more similar to ordinary Isabelle recursive definitions, providing also with nice syntax for mutual recursion. Translation take care automatically of the fact that, in contrast with Haskell, Isabelle requires patterns in case expressions to follow the order of datatype declarations. Carrying on with the example, we may consider the following code.

\begin{verbatim}
fn1 :: (a -> c) -> (b -> d) -> AC a b -> AC c d
fn1 f g k = case k of
  Ct x -> Ct (f x)
  Rt x y -> Rt (fn1 f g x) (fn2 f g y)
fn2 :: (a -> c) -> (b -> d) -> BC a b -> BC c d
fn2 f g k = case k of
  Dt x -> Dt (g x)
  St x y -> St (fn2 f g x) (fn1 f g y)
\end{verbatim}

This translates to HOLCF as follows.

\begin{verbatim}
consts
fn1 :: ('a :: pcpo ->' c :: pcpo) -> ('b :: pcpo ->' d :: pcpo) ->
  ('a,'b) AC -> ('c,'d) AC
fn2 :: ('a :: pcpo ->' c :: pcpo) -> ('b :: pcpo ->' d :: pcpo) ->
  ('a,'b) BC -> ('c,'d) BC
fixrec fn1 = (LAM f g k. case k of
  Ct·px1 => Ct·(f·px1) |
  Rt·px1·px2 =>
    Rt·(fn1·f·g·px1)·(fn2·f·g·px2))
and fn2 = (LAM f g k. case k of
  Dt·px1 => Dt·(g·px1) |
  St·px1·px2 =>
    St·(fn2·f·g·px1)·(fn1·f·g·px2))
\end{verbatim}

7 \textbf{HOL: Sentences}

Non-recursive definitions are treated in an analogous way as in the translation to HOLCF. Standard lambda-abstraction (\(\lambda\)) and function application are used here, instead of continuous ones. Partial functions, and particularly case expressions with incomplete patterns, are not allowed. The translation of terms (minus recursion and case expressions) may be summed up as follows.

\begin{verbatim}
[x :: a] = x_1 :: [a]
[c] = c_t
[x -> f] = \lambda x_t. [f]
[(a,b)] = ([a], [b])
[f a_1...a_n] = f_1 [a] ... [a_n]
\end{verbatim}

where \(f :: \tau, f_1 :: [\tau]\)

\begin{verbatim}
[let x_1 ... x_n in exp] = let [x_1] ... [x_n] in [exp]
\end{verbatim}
Recursive definitions set HOL and HOLCF apart. In HOL one has to pay attention to the distinction between primitive recursive functions (introduced by the keyword \texttt{primrec}) and generally recursive ones (keyword \texttt{recdef}). Termination is guaranteed for each primitive recursive function by the fact that recursion is based on the datatype structure of one of the parameters. In contrast, termination is not a trivial matter for recursion in general. A strictly decreasing measure needs to be provided, in association with the parameters of the defined function. This requires a degree of ingenuity that cannot be easily dealt with automatically. For this reason, we restrict the translation to HOL to primitive recursive functions. Mutual recursion is allowed for under some additional restrictions — more precisely:

1) all the functions involved are recursive in the first argument;
2) recursive arguments are of the same type in each function.

As an example, the translation of mutually recursive functions of type \(a \to b, \ldots a \to d\), respectively, introduces a new function of type \(a \to (b \times \ldots \times d)\) which is recursively defined, for each case pattern, as the product of the values correspondingly taken by the original ones. The following gives an example.

\[
\begin{align*}
fn3 & : \ AC\ a\ b \to (a \to a) \to AC\ a\ b \\
fn3 \ k\ f &= \ \text{case} \ k \ \text{of} \\
& \qquad Ct\ x \to \ Ct\ (f\ x) \\
& \qquad Rt\ x\ y \to \ Rt\ (fn4\ x)\ y \\
fn4 & : \ AC\ a\ b \to AC\ a\ b \\
fn4 \ k &= \ \text{case} \ k \ \text{of} \\
& \qquad Rt\ x\ y \to \ Rt\ (fn3\ x\ (\lambda z\ z))\ y \\
& \qquad Ct\ x \to \ Ct\ x
\end{align*}
\]

The translation to HOL of these two functions gives the following.

\[
\begin{align*}
\text{consts} \\
fn3 & : \ ('a :: type, 'b :: type)\ AC \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a, 'b)\ AC \\
fn4 & : \ ('a :: type, 'b :: type)\ AC \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a, 'b)\ AC \\
fn3_{-X}fn4_{-X} & : \ ('a :: type, 'b :: type)\ AC \Rightarrow \\
& \qquad (('aXX1 :: type \Rightarrow 'aXX1) \Rightarrow ('aXX1,'bXX1 :: type)\ AC)\* \\
& \qquad (('aXX2 :: type \Rightarrow 'aXX2) \Rightarrow ('aXX2,'bXX2 :: type)\ AC)
\end{align*}
\]
defs

\[ fn_3 \text{def : } fn_3 \equiv \lambda k. f. \text{fst } ((fn_3 \times fn_4 \times X) ::
\langle 'a :: type, 'b :: type \rangle \Rightarrow \langle (a \Rightarrow 'a) \Rightarrow \langle (a, b') \Rightarrow (unit, unit) \Rightarrow (unit, unit) AC \rangle \rangle k) \]

\[ fn_4 \text{def : } fn_4 \equiv \lambda k. f. \text{snd } ((fn_3 \times fn_4 \times X) ::
\langle 'a :: type, 'b :: type \rangle \Rightarrow \langle (unit \Rightarrow unit) \Rightarrow (unit, unit) AC \rangle *
\langle a \Rightarrow 'a' \Rightarrow (a, b') AC \rangle k) \]

primrec

\[ fn_3 \times fn_4 \times X (Ct \ pX_1) = (\lambda f. Ct f pX_1),
\lambda f. Ct pX_1) \]
\[ fn_3 \times fn_4 \times X (Rt \ pX_1 \ pX_2) =
(\lambda f. Rt (\text{snd } (fn_3 \times fn_4 \times X pX_1 f) pX_2),
\lambda f. Rt (\text{fst } (fn_3 \times fn_4 \times X pX_1 f) pX_2) \]

One may note that the type of the recursive function, for each of its
call in the body of non-recursive definitions, is given by instantiations
where the Isabelle unit type is replaced for each type variable which
is not occurring on the right hand-side, i.e. for each variable which is
not constrained by the type of the defined function. This is required by
Isabelle, in order to avoid definitions from which inconsistencies could
be derived. Other meta-level features are essentially common to both
translation.

8 Patterns

Multiple function definitions based on top level pattern matching are
translated as definitions based on case expressions. In fact, conversion
to case expressions makes it easier to deal with definitions that have
patterns in more than one variable, in HOL as well as in HOLCF.

data TV = F | T

ctlx :: TV \rightarrow TV \rightarrow TV

ctlx F a F = a
ctlx T a F = F
ctlx F a T = T
ctlx T a T = a

As an example, the above translates to HOLCF as follows.

domain TV = F | T

cconsts ctlx :: TV \rightarrow TV \rightarrow TV \rightarrow TV
Support for patterns in definitions and case expressions is more restricted in Isabelle than in Haskell. Nested patterns are overall disallowed. In case expressions, the case term is required to be a variable. Both of these restrictions apply to our translations. A further Isabelle limitation — sensitiveness to the order of patterns in case expressions — is dealt with automatically. Similarly, wildcards, not available in Isabelle, are dealt with and may be used, in case expressions as well as in function definition, though not in nested position. The translation to HOLCF can also handle incomplete patterns, also not allowed by Isabelle, in function definitions as well as in case expressions, by using \( \bot \) as default value. Neither guarded expressions and list comprehension are covered; these can be replaced quite easily anyway, using conditional expressions and \textit{map}.

\section{Classes}
Conceptually, type classes in Isabelle and Haskell are different things. The former are associated with sets of axioms, whereas the latter come with sets of function declarations. Moreover, Isabelle allows only for classes with a single type parameter. Most importantly, Isabelle does not allow for type constructor classes. The last limitation is rather serious, since it makes hard to cope with essential Haskell features such as monads and the \textit{do} notation. In contrast with the method proposed in [HMW05], we get partially around the obstacle by relying on an extension of Isabelle based on theory morphism (see section 11).

Defined classes are translated to Isabelle as classes with empty axiomatization. Every class is declared as a subclass of \textit{type}, also in the case of HOLCF — this is in order to allow for instantiation with lifted built-in types. Our translations cover only classes with no more than one type parameter. Instance declarations are translated to corresponding ones in Isabelle. Isabelle instances in general require proofs that class axioms are satisfied by the types, but as long as there are no axioms proofs are obviously trivial and carried out automatically. Method declarations are translated to independent function declarations with appropriate class annotation on type variables. Method definitions associated with instance declarations are translated to overloaded function definitions by using type annotation. An example follows.

\begin{verbatim}
axclass K a where
bsm :: a \rightarrow\ Bool
\end{verbatim}
The translation of this code to HOLCF gives the following.

```lean
axclass K < type
instance AC :: (\{pepo,K\}, \{pepo,K\}) K
by intro_classes

consts
  bsm :: 'a :: \{K,pepo\} \to tr
  stm :: 'a :: \{K,pepo\} \to tr
  default-bsm :: 'a :: \{K,pepo\} \to tr
  default-stm :: 'a :: \{K,pepo\} \to tr

defs
  AC-bsm_def : bsm ::
                 (\{K,pepo\},\{K,pepo\}) AC \to tr
                 == \lambda x. case x of
                   Ct \cdot pX \Rightarrow TT 
                 | Rt \cdot pX2 \cdot pX1 \Rightarrow FF

  AC-stm_def : stm ::
               (\{K,pepo\},\{K,pepo\}) AC \to tr
               == default-stm
```

Additional functions declared as default in method definition reflect an internal feature of the Programatica representation. Translating to HOL, on the other hand, we get the following.

```lean
axclass K < type
instance AC :: (\{type,K\}, \{type,K\}) K
by intro_classes

consts
  bsm :: 'a :: \{K,type\} \Rightarrow bool
  stm :: 'a :: \{K,type\} \Rightarrow bool
  default-bsm :: 'a :: \{K,type\} \Rightarrow bool
  default-stm :: 'a :: \{K,type\} \Rightarrow bool

defs
  AC-bsm_def : bsm ::
                 (\{K,type\},\{K,type\}) AC \Rightarrow bool
                 == \lambda x. case x of
                   Ct \cdot pX \Rightarrow True 
                 | Rt \cdot pX2 \cdot pX1 \Rightarrow False

  AC-stm_def : stm ::
               (\{K,type\},\{K,type\}) AC \Rightarrow bool
               == default-stm
```
In the internal representation of Haskell given by Programatica, function overloading is handled by means of dictionary parameters [Jon93]. This means that each function has additional parameters for the classes associated to its type variables. In fact, dictionary parameters are used to decide, for each instantiation of the function type variables, how to instantiate the methods called in the function body. On the other hand, overloading in Isabelle is obtained by adding type annotation to function definitions. Dictionary parameters are eliminated by the translations.

10 Built-in classes

Translation of built-in classes may involve giving some axioms and proving some lemmas. This is the case for equality, in particular. An HOLCF formalisation, based on the methods specification in [PJ03], can be given using axiomatic classes, as follows.

consts

heq :: 'a ⇒ 'a ⇒ bool
hneg :: 'a ⇒ 'a ⇒ bool

axclass Eq < pcpo

eqAx : ¬ (p = ⊥) ∧ ¬ (q = ⊥) ⟹ (heq p q) = ¬ (hneg p q)

Functions heq and hneg can be defined, for each instantiating type, using the translation of equality and inequality, respectively. In the case of lifted HOL types, we need to associate the HOL type to a syntactic class in order to carry out the instantiation — so for \( tr \) as well.

axclass Eq_bool < type

instance bool :: Eq_bool

by intro_classes

defs

\[ tr\_heq\_def : heq (p :: (\textit{'}a :: Eq\_bool) lift) q \equiv \neg (p = \bot) \land \neg (q = \bot) \land (p = q) \]\n
\[ tr\_hneg\_def : hneg (p :: (\textit{'}a :: Eq\_bool) lift) q \equiv \neg (p = \bot) \land \neg (q = \bot) \land \neg (p = q) \]

The instantiation requires a proof that the definitions satisfy the class axioms — here eqAx. In general, these proofs cannot be automated — the translation will print \textit{sorry} (a form of ellipsis in Isabelle) where the user needs to put in actual proofs. In the case of \( tr \), the following will do.

instance lift (Eq\_bool) Eq

apply (intro\_classes, unfold tr\_heq\_def tr\_hneg\_def)

apply auto

The corresponding HOL axiomatisation is of course less than satisfactory: it requires, as an extra-logical assumption, a restriction to
terminating programs.

\begin{verbatim}
consts
  heq :: ('a :: Eq) => 'a => bool
  hneq :: ('a :: Eq) => 'a => bool
axclass Eq < type
  axEq : heq p q == ~ hneq p q
\end{verbatim}

A general advantage of the axiomatic class approach is the guarantee of consistency. In contrast, when we added straightforward axioms, the resulting theory might not be conservative with respect to methods definition.

## 11 Monads

A monad is a type constructor with two operations — *eta*, injective, and *bind*, associative, with *eta* as left and right unit [Mog89]. The pseudo-imperative *do* notation in Haskell is based on monads; the list type constructor and *Maybe* are therein monads, as well. Isabelle, however, does not have type constructor classes — hence the problem of translating monadic operators adequately. Our approach to this problem is to rely on theory morphism as implemented in the AWE extensions [BJL06] to Isabelle. The AWE extensions assist in the construction of such theory morphisms by suitable Isar extensions and e.g. matching axioms to theories automatically.

A theory morphism is a map between theories which essentially maps signatures to signatures, axioms to theorems and theorems to theorems, preserving operations and arities. Theory morphisms allow theorems to be moved between theories by translating their proof terms. We can use this to implement parameterisation at the theory level. A *parameterised theory* $Th$ has a subtheory $Th_P$ which is the parameter. The parameter can contain axioms, constants and type declarations. It is instantiated by constructing a theory morphism into an instantiating theory $I$. We can now translate the proofs made in the abstract setting of $Th$ to the concrete setting of $I$, resulting in a new theory (see [BJL06] for details). We use this facility to implement a notion of monads in Isabelle. On an abstract level, we declare a unary type constructor $M$, the two monad operations and the relevant axioms. This gives rise to a hierarchy of theories, culminating in *Monad*. To show that a specific type constructor forms a monad, we have to construct a theory morphism from the theory $MonadAxms$; this entails defining the monad operations and showing the monad axioms. This is done interactively. We now consider an example of translation to HOL which uses *Monad*.

\begin{verbatim}
data LS a = N | C a (LS a)
instance Monad LS where
  return x = C x N
  x >>= f = case x of
    N -> N
    C a b -> cnc (f a) (b >>= f)
\end{verbatim}
\textit{cnc} :: \textit{LS} \, a \rightarrow \textit{LS} \, a \\
\hspace{1cm} \textit{cnc} \, x \, y = \text{case} \, x \, \text{of} \\
\hspace{1.5cm} N \rightarrow y \\
\hspace{1.5cm} C \, w \, z \rightarrow \textit{cnc} \, z \, (C \, w \, y)

These definitions translate to HOL as follows.

\textit{datatype} \ 'a \textit{LS} = N \mid C \, 'a \ (\textit{LS})
\textit{consts}
\hspace{1cm} \text{\textit{return\_LS}} :: 'a \Rightarrow \textit{LS}
\hspace{1cm} \text{\textit{bind\_LS}} :: \textit{LS} \Rightarrow ('a \Rightarrow 'b \textit{LS}) \Rightarrow 'b \textit{LS}
\hspace{1cm} \text{\textit{cnc}} :: \textit{LS} \Rightarrow \textit{LS} \\
\textit{defs}
\hspace{1cm} \text{\textit{return\_LS\_def}} : \text{\textit{return\_LS}} :: ('a \Rightarrow 'a \textit{LS}) = \lambda x. C \, x \, N
\hspace{1cm} \text{\textit{primrec}}
\hspace{1.5cm} \text{\textit{bind\_LS}} \, N = \lambda f. N
\hspace{1.5cm} \text{\textit{bind\_LS}} \, (C \, pX \, 1 \, pX \, 2) = \lambda f. \textit{cnc} \, (f \, pX \, 1) \, (\textit{bind\_LS} \, pX \, 2 \, f)
\hspace{1cm} \text{\textit{primrec}}
\hspace{1.5cm} \text{\textit{cnc}} \, N = \lambda b. b
\hspace{1.5cm} \text{\textit{cnc}} \, (C \, pX \, 1 \, pX \, 2) = \lambda b. \textit{cnc} \, pX \, 2 \, (C \, pX \, 1 \, b)

To build up the instantiation of \textit{LS} as a monad, we define a mapping \textit{m\_LS} from the theory \textit{MonadType} to the instantiating theory — here \textit{Tx} — which maps the generic single parameter operator \textit{M} to \textit{LS}.

\textit{thymorph} \textit{m\_LS} : \textit{MonadType} \rightarrow \textit{Tx}
\hspace{1cm} \text{maps} [(\textit{\textquotesingle a \textit{MonadType}.\textit{M}} \rightarrow \textit{\textquotesingle a \textit{Tx}.\textit{LS}})]

We can now instantiate the parameterised theory \textit{MonadOps} using \textit{m\_LS}, which gives us a declaration of instantiated methods. (Note how this builds up our current theory.)

\textit{t\_instantiate} \textit{MonadOps} \textit{mapping} \textit{m\_LS}

The instantiated methods are now defined, using the functions given by the translation from Haskell.

\textit{defs}
\hspace{1cm} \textit{LS\_eta\_def} : \text{\textit{eta}} = \textit{return\_LS}
\hspace{1cm} \textit{LS\_bind\_def} : \text{\textit{bind}} = \textit{bind\_LS}

To construct a mapping from the theory \textit{MonadAxms} to \textit{Tx}, we need to prove the monad axioms as HOL lemmas (in this case, by straightforward simplification).
lemma LS Unit : bind (eta x) t = t x
lemma LS Runtime : bind (t :: 'a LS) eta = t
lemma LS Assoc : bind (bind (s :: 'a LS) t) u =
bind s (λx. bind (t x) u)
lemma LS Eta Inj : eta x = eta y ⇒ x = y

Now, we can define a mapping from MonadAxms to Tx, and the in-
stantiate the theory Monad theory with it. This gives us access to the
theorems proven in Monad and e.g. the monadic syntax defined there.

\text{thymorph mon}\_\text{LS} : \text{MonadAxms} \rightarrow \text{Tx}

\text{maps} \left[\left('a \text{MonadType}.M ightarrow 'a \text{Tx}.LS\right)\right]
\left[\left(\text{MonadOpEta}.\eta \rightarrow \text{Tx}.\eta\right),\right.
\left(\text{MonadOpBind}.\text{bind} \rightarrow \text{Tx}.\text{bind}\right)\]

\text{t, instantiate Monad mapping mon}\_\text{LS}

The Monad theory allows for the characterisation of single parameter
operators. In order to cover other monadic operators, a possibility is to
build similar theories for type constructors of fixed arity. Moreover, an
approach altogether similar to the one shown for HOL can be used, in
principle, for translation to HOLCF.

12 Conclusion

We have shown how shallow embedding may be used to translate Haskell
to Isabelle higher-order logics. The main advantage of this approach is
to get as much as possible out of the automation currently available in
Isabelle, especially with respect to type checking. HOLCF in particular
provides with an expressive semantics covering lazy evaluation, as well
as with a smart syntax — also thanks to the \text{fixrec} package. The main
disadvantage, on the other hand, lies with the fact that Isabelle does not
have type constructor classes. In the case of HOL, we have shown how
it is possible to get around the obstacle, at least partially, by relying on
the axiomatic characterisation of monads and on a proof-reuse strategy
that actually minimises the need for interactive proofs.

Future work will use this framework for proving properties of Haskell
programs. For monadic programs, we plan to use the monad-based dy-
namic Hoare and dynamic logic that already have been formalised in
Isabelle [Wal05].

Our translation tool from Haskell to Isabelle is part of the
Heterogeneous Tool Set Hets and can be downloaded from
http://www.dfki.de/sks/hets.

References

[ABB+05] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Ver-
ifying Haskell programs using constructive type theory. In
ACM-SIGPLAN 05, 2005.


A Probabilistic Model for Parametric Fairness in Isabelle/HOL *

Jinshuang Wang, Xingyuan Zhang, Yusen Zhang, and Huabing Yang

PLA University of Science and Technology, Nanjing 210007, China
wang_jinshuang@163.com xingyuanz@gmail.com zys49nj@126.com
yanghuabing@gmail.com

Abstract. In paper [1], a liveness proof method suitable for inductive protocol verification is proposed. The utility of this method has been confirmed by several machine checked formal verifications[2-4]. One remaining question about [1] is the meaning of Parametric Fairness, a new fairness notion adapted from Pnueli’s Extreme Fairness[5] to suit the setting of higher-order logic. This paper tries to answer this question. As a standard practice in establishing a fairness notion, this paper constructs a probabilistic model for parametric fairness. Using this model, it is shown that most infinite executions of a concurrent system are parametrically fair. Therefore the definition of parametric fairness in paper [1] is reasonable. This work gives a firmer basis for existing and forthcoming formal verifications based on the method of paper [1].

Keywords: Liveness Proof, Inductive Protocol Verification, Probabilistic Model, Parametric Fairness.

1 Introduction

In paper [1], an extension to Paulson’s inductive protocol verification approach[6-8] is proposed to deal with liveness properties. The practicability of this extension has been shown by several machine checked formal verifications[2-4], where the liveness properties of elevator control systems and mobile Ad Hoc network protocols are proved. The method of paper [1] is based on a new notion of fairness named parametric fairness. Parametric fairness is an adaption of extreme fairness[5] to suit the setting of higher-order logic. Extreme fairness is a nonstandard fairness notion introduced to simplify liveness proofs.

An execution trace $\sigma$ is said to be extremely fair if it is fair with respect to every state predicate $\varphi$. A literal translation of the original extreme fairness definition into higher-order logic using universal quantification over state predicate variable $\varphi$ is problematic. For any infinite execution $\sigma$, we can construct a higher-order state predicate $\varphi_\sigma$ which is not fairly treated in $\sigma$. By instantiating variable $\varphi$ to state predicate $\varphi_\sigma$, it can be

---

* This research was funded by National Natural Science Foundation of China, under grant 60373068 ‘Machine-assisted correctness proof of complex programs'
shown that execution $\sigma$ is not fair. Therefore the universal quantification
over $\varphi$ makes the fairness definition so restrictive that no execution can
satisfy it.

An investigation into liveness proofs shows that every liveness proof re-
lies on some state predicates being fairly treated, but the number of such
state predicates is always finite. Therefore, the universal quantification
over $\varphi$, which requires every state predicate to be treated fairly, is not
necessary. As a solution, paper [1] requires that the state predicates used
in liveness proof be given explicitly as a parameter $pel$ to the expression
$\text{`PF cs pel } \sigma$. The parameter $cs$ is the underlying concurrent system. Al-
though this parametric formulation avoids the problematic derivation in
last paragraph, it remains to show that this new formulation is sufficiently
reasonable to agree with people’s intuition about concurrent execution,
which is explained as follows:

By execution, we mean an infinite sequence of events, where every event
is eligible to happen under the system state determined by all preceding
events in the sequence. The set of events eligible to happen under a state
is determined by the definition of the underlying computer system. A
computer system is concurrent if under each system state, there may be
more than one event eligible to happen. Concurrent execution is an
execution sequence of concurrent system, where the events in sequence
are chosen nondeterministically. The behavior of a concurrent system is
the set of its concurrent executions.

This paper shows that if the nondeterminacy in concurrent executions
is resolved by random choice, a probability space can be constructed
with the execution set as basis, and that the set of parametrically fair
executions is measurable and has probability 1. The results means that
almost all concurrent executions are parametrically fair and our definition
of parametric fairness is natural and reasonable.

Similar works have already been done for extreme fairness and other no-
tions of fairness[5,9,10], but all in different settings. Our contribution is
to show that these results still hold in the setting of higher-order logic,
under our notion of concurrent system and parametric fairness. Addition-
ally, our work is more reliable because all preceding works are pencil
and paper ones done outside object logic, while ours is inside the object
logic Isabelle/HOL, machine-checked and coexisting with concrete verifi-
cations. Parametric fairness has the same motivation as extreme fairness,
i.e. to reflect the probabilistic nature of concurrent executions. This paper
shows that the goal is indeed achieved. Therefore the liveness reasoning method proposed in [1] is now on a firmer basis.

The rest of this paper is organized as follows: Section 2 introduces the notion of parametric fairness. Section 3 introduces the notion of probabilistic execution and constructs a probability space on sets of infinite executions. Section 4 assigns a probabilistic meaning to parametric fairness by showing that the probability of the set of fair executions equals to 1. Section 5 summarizes related works. Section 6 concludes.

2 The notion of parametric fairness

2.1 Concurrent systems

In inductive approach, system states are identified with finite executions, which are represented as lists of events. These events are arranged in reverse order of happening, the decision of which event to happen next is decided according to current system state. Formal definitions for concurrent system is given in Fig. 1, where the type of events is a polymorphic type 'a, and the type of system states is 'a list. We identify system states with finite execution traces. The terms system state and finite execution are used interchangeably. Both of them are written as \( \tau, \tau', \tau_1, \tau_2 \), etc.

```
constdefs heads :: (nat => 'a) => 'a list => 'a (\_, [64, 64] 1000)
  \( \sigma_1 \in \sigma \)

consts prefix :: (nat => 'a) => 'a list => 'a list (\_, [64, 64] 1000)
  primrec [\sigma]_0 = []
  [\sigma]_n = \sigma_1 \# [\sigma]

constdefs may-happen ::
  'a list => \( \{ \text{a list} \times \text{a list} \} \) set => \( \text{'a list} \times \text{bool} \) (\_, [64, 64] 50)
  \( r [c=>] e \Xi (r', e) \in cs \)

consts vt :: \( \{ \text{a list} \times \text{a list} \} \) set => \( \text{'a list set} \times \text{inductive} \) vt cs
  intrcs
  vt-nil [] [nat] : [] \in vt cs
  vt-cons [vt] s [nat] : \_ \in vt cs; \_ [c=>] \_ \Rightarrow (e \# r) \in vt cs

consts derivable :: \( \text{a list} \Rightarrow \text{bool} \) (\_, [64, 64] 50)

def (overloaded)
  fst-valid-def cs \_ \tau \_ r \in vt cs
  in-valid-def cs \_ \sigma \_ \_ \[\sigma], [e> \_ \tau, \_ \_ \_ \_ \_ \_ \] \sigma_i
```

Fig. 1. The definitions of concurrent system

The type of infinite executions is nat => 'a, which is often abbreviated as 'a seq. For an infinite execution \( \sigma \), the event happens at the \( i-th \) step is \( \sigma_i \), which is also written as \( \sigma_i \). The first \( i \) events of an infinite execution
\( \sigma \) can be packed into a list, which exactly forms a finite execution. Such a packing is written as \([\sigma]_i\), which is also called a prefix of \( \sigma \).

A concurrent system is often written as \( cs \), and its type is \((\text{'a list} \times \text{'a})\) set. The expression \((\tau, e) \in cs\) means that the event \( e \) is legimate to happen under state \( \tau \) according to \( cs \). The notation \((\tau, e) \in cs\) is also written as \( \tau \downarrow cs \). The set of valid finite executions of \( cs \) is written as \( vt \downarrow cs \). The expression \( \tau \in vt \downarrow cs \) is also written as \( cs \vdash \tau \). The operator \( \vdash \) is overloaded, so that \( \sigma \) is a valid infinite execution of \( cs \) can be written as \( cs \vdash \sigma \). An infinite execution \( \sigma \) is valid under \( cs \) iff all of its prefixes are valid.

2.2 Embedding LTL

LTL (Linear Temporal Logic) is widely used for the specification and verification of concurrent systems. A shallow embedding of LTL is given in Fig. 2. In this paper, LTL is used to express various temporal properties of infinite executions, including liveness properties.

```
<table>
<thead>
<tr>
<th>types</th>
<th>'a tfl = {sat \rightarrow 'a} \rightarrow sat \rightarrow bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>consts</td>
<td>valid-under :: 'a \Rightarrow 'b \Rightarrow bool [: \Rightarrow [64, 66, 50]</td>
</tr>
<tr>
<td></td>
<td>defs (overloaded) ( pr \Vdash \varphi \equiv \text{let } (\omega, i) = pr \equiv \varphi ) i</td>
</tr>
<tr>
<td></td>
<td>defs (overloaded) ( \sigma \Vdash \varphi \equiv (\sigma: \text{sat} \Rightarrow \lambda, (0:\text{sat})) \Vdash \varphi</td>
</tr>
<tr>
<td></td>
<td>constdefs always :: 'a tfl \Rightarrow 'a tfl ([\boxdot [64, 66])</td>
</tr>
<tr>
<td></td>
<td>( \Box \varphi \equiv \lambda \sigma i. \forall j. i \leq j \rightarrow (\sigma, j) \Vdash \varphi</td>
</tr>
<tr>
<td></td>
<td>constdefs eventually :: 'a tfl \Rightarrow 'a tfl ([\eventually [64, 66])</td>
</tr>
<tr>
<td></td>
<td>( \Diamond \varphi \equiv \lambda \sigma i. \exists j. i \leq j \land (\sigma, j)) \Vdash \varphi</td>
</tr>
</tbody>
</table>
```

Fig. 2. A shallow embedding of LTL.

The type of LTL formulae is defined as \( 'a tfl \). The expression \((\sigma, i) \models \varphi \) means that LTL formula \( \varphi \) is valid at the \( i \)-th step of \( \sigma \). The operators \( \text{always} \Box \) and \( \text{eventually} \Diamond \) are defined literally.

2.3 The introduction of parametric fairness

In paper [1], two liveness proof rules are derived. The one for response properties is:

\[ [\text{RESP} \ cs \ F \ E \ N \ P \ Q; \ cs \vdash \sigma; \ PF \ cs \ \|F, E, N\| \sigma] \rightarrow [\sigma \models \Box(\psi) \rightarrow \Diamond(\varphi)] \]

and the one for reactivity properties is:

\[ [\text{REACT} \ cs \ F \ E \ N \ P \ Q; \ cs \vdash \sigma; \ PF \ cs \ \|F, E, N\| \sigma] \rightarrow [\sigma \models \Diamond(\psi) \rightarrow \Box(\varphi)] \]

To prove a liveness result, an execution path must be found, which goes from the starting state characterized by \( \psi \) to the ending state characterized by \( \varphi \). Such a path is represented by a list of (state predicate, event
function)-pairs, which needs to be fairly treated, so that the execution of the concurrent system will eventually go along the path. The path is abstract in the sense that instead of concrete (state, event)-pairs, it consists of (state predicate, event function)-pairs, where the state predicate specifies the condition under which the event yielded by the event function is to happen. The functions \(F\) and \(E\) are given by verification staff to generate the state predicates and event functions respectively. The \(N\) is the length of the path. The expression \(\{F, E, N\}\) represents the generated (state predicate, event function)-pair list. The pair list should be properly designed to form a chain leading from \(\psi\) to \(\varphi\). Such a requirement is expressed in the definitions of both \(RESP\) and \(REACT\).

\(RESP\) and \(REACT\) contain some additional requirements peculiar to response and reactivity properties respectively, the explanation of which is irrelevant for the purpose of this paper. Interested readers may consult [1] for details.

Anyway, these two premises will be resolved to obtain liveness results of the form:

\[
\text{cs} \vdash \sigma; \; PF\; cs\; \{F, E, N\}; \; \sigma \iff \sigma \vdash \Box(\psi) \rightarrow \Diamond(\varphi)
\]

or of the form:

\[
\text{cs} \vdash \sigma; \; PF\; cs\; \{F, E, N\}; \; \sigma \iff \sigma \vdash \Box(\psi) \rightarrow \Box(\varphi)
\]

The \(cs \vdash \sigma\) condition requires the infinite execution \(\sigma\) be a valid execution of concurrent system \(cs\). The purpose of this paper is to give a probabilistic meaning to the remaining \(PF\; cs\; \{F, E, N\}; \; \sigma\), which requires that the infinite execution \(\sigma\) is parametrically fair with respect to the parameter \(\{F, E, N\}\). The formal definition of \(PF\) is given in Fig. 3.

<table>
<thead>
<tr>
<th>EF</th>
<th>((\text{a list } \times \text{'a})) set (\Rightarrow (\text{a list } \Rightarrow \text{bool}) \Rightarrow (\text{a list } \Rightarrow \text{'a}) \Rightarrow (\text{nat } \Rightarrow \text{'a}) \Rightarrow \text{bool} \rangle )</th>
<th>(E)</th>
<th>((\text{a list } \times \text{'a})) set (\Rightarrow (\text{nat } \Rightarrow \text{'a}) \Rightarrow \text{bool} \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c s ; P ; E ; \sigma \equiv )</td>
<td>(\sigma \vdash \Box(\text{nat } \Rightarrow \text{bool} \rangle ; \wedge ; \sigma_i = E ; \sigma))</td>
<td>(E)</td>
<td>((\text{a list } \times \text{'a})) set (\Rightarrow (\text{nat } \Rightarrow \text{'a}) \Rightarrow \text{bool} \rangle )</td>
</tr>
<tr>
<td>types (\text{a pe} = (\text{a list } \Rightarrow \text{bool}) \times (\text{a list } \Rightarrow \text{'a}))</td>
<td>(PF)</td>
<td>((\text{a list } \times \text{'a})) set (\Rightarrow (\text{a pe} ; \text{list} \Rightarrow (\text{nat } \Rightarrow \text{'a}) \Rightarrow \text{bool} \rangle )</td>
<td></td>
</tr>
</tbody>
</table>
remedy, which only requires the (state predicate, event function)-pairs appearing in parameter pel being fairly treated. In the following, we are going to construct a measure space over the power set of all infinite executions, and show that almost all infinite executions satisfy the requirement of PF. However, since our approach works at meta level, the premise PF cs \{F, E, N\} σ can not be removed from the final results. It is there to provide people with information that the liveness result is achieved through the enabling execution path \{F, E, N\}.

When a complex concurrent system is verified, people usually need to derive many liveness results, each with its own enabling path. The result in this paper also means that the requirement of PF cs pel σ, using the conjunction of all these enabling paths as the parameter pel, can still be satisfied by almost all infinite executions.

3 Probability space construction

3.1 Formalizing probabilistic execution

To model random choice, we introduce a function \(R\), where \(R(\tau, e)\) is the probability of event \(e\) being chosen to happen under system state \(\tau\). Therefor, \(R(\tau, e) = 0\) means event \(e\) is not eligible to happen under state \(\tau\), while \(R(\tau, e) > 0\) means \(e\) is eligible to happen with \(R(\tau, e)\) as the happening probability. Every \(R\) represents an execution strategy of a concurrent system. The underlying concurrent system can be defined in terms of \(R\) as \(CS \equiv \{(\tau, e), 0 < R(\tau, e)\}\). The set of events eligible to happen under state \(\tau\) is given as \(N \tau \equiv \{e, 0 < R(\tau, e)\}\).

It is natural to assert the following axioms:

1. \(0 \leq R(\tau, e) \land R(\tau, e) \leq 1\), which is a routine requirement of probability theory.
2. \(CS \vdash \tau \rightarrow (\sum e \in N \tau. R(\tau, e)) = 1\). The purpose of this axiom is to fulfill the standard probability theory requirement that the summation of all possible outcomes must equal to 1. This axiom also entails that every valid finite execution can always be extended by at least one event. By introducing a Tick event, which represents the ticks of a system wide clock, this requirement can be satisfied easily, since nothing can prevent time from advancing.
3. \(\forall \tau e. 0 < R(\tau, e) \rightarrow bnd \leq R(\tau, e)\), where \(bnd\) is the lower bound of all nonzero \(R\)-probabilities. It is natural to require that \(0 < bnd \land bnd < 1\). This axiom also entails finite \((N \tau)\), which means that the underlying concurrent system \(CS\) is finitely branching,
Funcion $R$ induces a measure function $\pi$ on finite executions:

$$\pi [] = 1$$
$$\pi (c \# r) = R(r, c) * \pi r$$

For any valid system state $r$, we have $\pi r > 0$. For any valid infinite execution $\sigma$, we have $\forall i. \pi [\sigma]_i > 0$. The base set (or sample space), on which we are going to construct the probability space, is defined as:

$$Path \equiv \{ \sigma. (\forall i. \pi [\sigma]_i > 0) \}$$

It can be shown that $(CS \vdash \sigma) = (\sigma \in Path)$, i.e., $Path$ coincides with the set of all valid infinite executions of the underlying concurrent system $CS$.

3.2 Outline of the construction

The definition of probability space is given in Fig. 4. This is a rather standard definition, where a probability space is defined to be a measure space $(U, F, Pr)$, where $U$ is the base set, $F$ the family of measurable sets (also called measurable), $Pr$ the measure function. For a probability space $(U, F, Pr)$, the measure of the base set must be 1. The definition of measure space uses standard notions such as $\sigma$-algebra, positivity and countable additivity. In the definition of countable additivity, we use Isabelle library function $\text{sums}$, where ‘$f \text{sums} c$’ stands for $\sum_{n=0}^\infty f(n) = c$.

The rest of the definitions are self-explaining.

The notion of $\sigma$-algebra is a generalization of algebra. As shown in Fig. 4, the only difference is that algebra is closed under finite union, while $\sigma$-algebra is closed under countable union. A standard way to obtain $\sigma$-algebra is to construct an algebra $(U, F)$ first, and then use operator $\text{sigma}$ to generate a $\sigma$-algebra $(U, \text{sigma}(U,F))$. The definition of $\text{sigma}$ is:

```plaintext
consts sigma :: ('a set × 'a set set) ⇒ 'a set set
inductive sigma M intro
  basic: (let $(U, A) = M \in (a \in A) \implies a \in \text{sigma} M$
  empty: ({} \in \text{sigma} M
  complement: a \in \text{sigma} M \implies (let $(U, A) = M \in U - a) \in \text{sigma} M$
  union: $(\forall i: \text{nat}, a i \in \text{sigma} M) \implies (\bigcup i, a i) \in \text{sigma} M$
```

According to Carathéodory’s extension theorem[11, 12], a measure $Pr$ on $F$ can be generalized naturally to a measure $Pr’$ on $\text{sigma}(U,F)$, so that $(U, \text{sigma}(U,F), Pr’)$ forms a measure space. The presentation of this extension theorem in Isabelle/HOL is as follows:

```plaintext
[algebra $(U, F)$; positive $(F, Pr)$; countably-additive $(F, Pr)$]
⇒ \exists P. (\forall A. A \in F \implies P A = Pr A) \land measure-space $(U, \text{sigma} (U, F), P)$
```
This theorem suggests the way we are going to construct the probability space for infinite executions. The base set is Path, on top of which a subsets family PA is defined such that algebra(Path, PA) holds. A measure \( \mu \) on \( PA \) is given with \( \mu(\text{Path})=1 \). Using extension theorem, we get a measure \( P \) on sigma(Path, PA) which satisfies \( P(\text{Path})=1 \) and measure-space(Path, sigma(Path,PA), P). It follows immediately that \( \text{Path, sigma(Path,PA), P) is a probability space.} \)

### 3.3 An algebra of infinite execution sets with measure \( \mu \)

The set of infinite executions with finite execution \( \tau \) as prefix is defined as follows:

\[
palg\emph{-emb} \tau \equiv \{ \sigma \in \text{Path}. \left[\sigma\right]_{\tau} = \tau \}
\]

The \( |\tau| \) in this definition is the length of \( \tau \).

A set of infinite executions is said to be supported by a list of finite executions \( \{\tau_0, \tau_I, \tau_2, \ldots, \tau_n\} \) (or, equivalently, supported by finite set \( \{ \tau_0, \tau_I, \tau_2, \ldots, \tau_n\} \)), if every infinite execution in the set is prefixed by some \( \tau_i \) among \( \tau_0, \tau_I, \tau_2, \ldots, \tau_n \). It is easy to see that the set supported by \( \{\tau_0, \tau_I, \tau_2, \ldots, \tau_n\} \) is the \( \text{un} \) (palg\emph{-emb} \( \tau_i \)). But we write the supported set slightly different as palgebra\emph{-emb}([\( \tau_0, \tau_I, \tau_2, \ldots, \tau_n \)], which is defined as the following:
\textbf{consts} \textit{palgebra-embed :: 'a list \Rightarrow 'a seq set} \\
\textbf{primes} \\
\textit{palgebra-embed} [] = {}
\textit{palgebra-embed} (r \#l) = (palg-embed \ r) \cup \textit{palgebra-embed} \ l \\

The subset family \textit{PA} mentioned earlier is defined as:
\[ \textit{PA} \equiv \{ S. \ \exists \ l. \ \textit{palgebra-embed} \ l = S \land S \subseteq \textit{Path} \} \]

Notice \( l \) in this definition is of the form \([\tau_0, \tau_1, \tau_2, \ldots, \tau_n] \}. Therefore, the set family \textit{PA} can be understood as consisting of only those sets of valid infinite executions which are supported by some list of finite executions.

We intend to base the measure of a set \( S \in \textit{PA} \) on the measures of its supporting lists. For this purpose, the measure \( \mu_0 \) is defined as:
\[ \mu_0 \ l \equiv (\sum \tau \in \text{set \ l}. \ \pi \ \tau) \]

However, a set \( S \) may be supported by many different lists with different \( \mu_0 \)-values. Suppose \( l \) is a supporting list of \( S \), \( \tau_1 \) is a valid finite execution contained in \( l \), \( \tau_2 \) is another valid finite execution with \( \tau_1 \) as suffix, we have \( S = \textit{palgebra-embed}(l) = \textit{palgebra-embed}(\tau_2\#l) \), i.e. \( S \) is supported by both \( l \) and \( \tau_2\#l \). However, since \( \mu_0(\tau_2\#l) = \pi(\tau_2) + \mu_0(l) \), we have \( \mu_0(l) < \mu_0(\tau_2\#l) \). Since we can repeat such constructions for any supporting list \( l \), there seems to be no upper limit for the \( \mu_0 \)-values of the supporting lists. It is natural to define the measure of \( S \) to be the lower limit of these \( \mu_0 \)-values. The following measure \( \mu_1 \) is a refinement of \( \mu_0 \) to serve this intention:
\[ \mu_1 \ l \equiv \inf (\lambda r. \ \exists \ l'. \ \textit{palgebra-embed} \ l = \textit{palgebra-embed} \ l' \land \mu_0 \ l' = r) \]

Measure \( \mu_1 \) will always return the measure of \( S \) properly, no matter which \( l \) among the supporting lists of \( S \) is used as argument.

Due to the restriction of Isabelle/HOL, we cannot give a measure on \textit{PA} explicitly. To solve this problem, the following measure \( \mu \) is defined on the type of all infinite execution sets, not just those in \textit{PA}:
\[ \textbf{consts} \ \mu :: 'a \ \textit{seq set} \ \Rightarrow \ \textit{real} \]
\[ \mu S \equiv \sup (\lambda r. \ \exists b. \ \mu_1 b = r \land (\textit{palgebra-embed} \ b) \subseteq S) \]

It can be shown that \([S \in \textit{PA}; \ S = \textit{palgebra-embed}(l)] \Rightarrow \mu(S) = \mu_1(l) \], which means the formal definition of \( \mu \) is in accordance with our informal intentions discussed above. The definitions of \( \mu \) and \( \mu_1 \) are copied from

It can be proved that \textit{algebra} (\textit{Path}, \textit{PA}). However, \textit{algebra} (\textit{Path}, \textit{PA}) is not rich enough to accommodate the set of fair executions. We are going to extend it to a \( \sigma \)-algebra using the \textit{Carathéodory's extension theorem} with \( U \rightarrow \textit{Path}, F \rightarrow \textit{PA}, \ Pr \rightarrow \mu \). For this purpose, we proved that
positive \((PA, \mu)\) and countably-additive \((PA, \mu)\) hold. By applying the extension theorem, we obtain a measure \(P\) with \(S \in PA \implies P(S)=\mu(S)\) and measure-space \((Path, \sigma(Path, PA), P)\). Finally, we get the probability space \((Path, \sigma(Path, PA), P)\), on which we are going to prove that the set of parametrically fair executions is measurable and has probability 1.

The proof of algebra \((Path, PA)\) is straightforward and skipped. The proof of countably-additive \((PA, \mu)\) is based on the fact that:

\[
\{range \ f \subseteq PA: \forall n. \ m \neq n \implies f_m \cap f_n = \{}\}, \ (\bigcup f_n) \in PA
\implies \exists N. \forall n. \ N \leq n \implies f_n = \{}
\]

The proof of this lemma is done using an argument similar to the proof of König's Lemma[12]. This lemma means that for any family \(f\) of subsets with \((\bigcup i. f_i) \in PA\), there are only finite many \(f(i)\)s that are nonempty. Therefore, countable additivity is reduced to finite additivity, which is proved easily by induction.

Some routine properties of probability space are listed as below:

- Monotonicity: If \(A\) and \(B\) are measurable, then \(P(A) \leq P(B)\).
- Subadditivity: If family \(A_n (n \in \omega)\) is measurable, then \(P(\bigcup n. A_n) \leq \sum n. P(A_n)\).
- Lower Continuity: If \(A_0 \subseteq A_1 \subseteq \ldots \subseteq A_n \ldots (n \in \omega)\) is a chain of measurable, then \((\lambda n. P(A_n)) \rightarrow P(\bigcup A_n)\). The expression \((\lambda n. f(n)) \rightarrow c\) is Isabelle/HOL notation for \(\lim_n f(n) = c\).
- If \(\forall i. P(f(i)) = 1\), then \(P(\bigcap i. f(i)) = 1\).

4 The probabilistic meaning of parametric fairness

The purpose of this section is to show that:

\[
\text{set pcl} \neq \{} \implies P \{\sigma \in Path. PF CS pcl \sigma\} = 1
\]

which means the probability of the set of valid parametrically fair executions is 1. This result gives a probabilistic meaning to parametric fairness by showing that almost all valid probabilistic executions are parametrically fair. All the sets, which appear as arguments to \(P\) in this section, have been proved to be measurable in the probability space constructed in Section 3.

For this purpose, the following lemma is proved:

\[
\text{set pcl} \neq \{} \implies \{\sigma \in Path. PF CS pcl \sigma\} = (\bigcap_{l \in \text{set pcl}} \text{Fair}_l \cap CS l)
\]
to reduce the calculation of $P\{\sigma \in \text{Path}, PF CS pel \sigma\}$ to the calculation of $P(\bigcap_{i \in \text{set} \ ts} \text{Fair}\ i \ CS \ iff)$, which is relatively easier to handle. The Fair represents a general way to present fairness[10], where fairness is defined relative to labels. The definition of Fair is given in Fig. 5.

![Fig. 5. A general fairness notion](image)

The definition of Fair uses fair$, where fair$i \ i cs \ \gamma \ \sigma$ means the particular execution$\sigma$ is fair to label$i$. An infinite execution is said to be fair to a label$i$ if$i$ is taken infinite many times whenever it is enabled infinite many times in$\sigma$. The parameter$\gamma$ in both Fair$i \ i cs \ \gamma$ and fair$i \ i cs \ \gamma$ is used to assign label sets to events so that the notions enabled and taken can be made precise. For any event$e$, expression$\gamma(\tau, e)$ represents the set of labels assigned to$e$ under system state$\tau$. A label$i$ is said to be enabled in state$\tau$(written as enabled$i \ i cs \ \gamma \ \tau$) if there exists some event$e$ eligible to happen under state$\tau$ and$i$ belongs to$\gamma(\tau, e)$. A label$i$ is said to be taken in state$\tau$ if$i$ is assigned to the last execution step of$\tau$. The last step of$\tau$ is denoted by$\text{ld}(\tau)$and the state before the last step is denoted by$\text{tl}(\tau)$.

The expression Fair$i \ i cs \ \gamma$ yields the set of infinite executions of concurrent system$cs$, which are fair to label$i$. The expression$\bigcap_{i \in \text{set} \ ts} \text{Fair}\ i \ CS \ iff$represents executions of$CS$which are fair to every label$i$in the set$\text{set}(pel)$. In the latter expression, labels are the$(P, E)$-pairs contained in the list$pel$. A$(P, E)$-pair is said to be assigned to$(\tau, e)$if$P(\tau) \land e = E(\tau)$holds. This notion is reflected in the function$\text{If}$, which is used as the parameter$\gamma$.

The next step is to show that $P(\bigcap_{i \in \text{set} \ ts} \text{Fair}\ i \ CS \ iff) = 1$. Since we have $P(\bigcap_{i} f \ i) = 1 \implies P(\bigcap_{i} f \ i) = 1$ for any set family$f$, it suffices to show $P(\text{Fair}\ i \ CS \ iff) = 1$ for any one label$i$. Equivalently, it is to show that $P(\text{Path} - (\text{Fair}\ i \ CS \ iff)) = 0$. Notice the$i$here is a$(P, E)$-pair. The next lemma we have proved is:

$$Path - (\text{Fair}\ i \ CS \ iff) = \bigcup_{\tau \in \{xs, CS \vdash xs\}} \Gamma \ i CS \ iff$$
The idea behind this lemma is that the unfair set $\text{Path} - (\text{Fair} \land cs \gamma)$ can be divided into a family of subsets, each of which is indexed by a finite execution $\tau$. Now, we only need to show:

$$P(\bigcup_{\tau \in \{xs. \text{CS} \vdash xs\}} \Gamma \land \text{CS} \land \tau) = 0$$

According to the subadditivity of probability measure mentioned in Section 3.3, it is sufficient to show that $P(\Gamma \land \text{CS} \land \tau) = 0$ for each finite execution $\tau$. The definition of $\Gamma$ is:

$$\Gamma \land \text{CS} \land \gamma \land \tau \equiv \\
\{\sigma \in \text{poly} \cdot \text{embed} \tau . \land \land \forall i \exists j. i \leq j \land \text{enabled} \land \text{cs} \land \gamma [\sigma]_j + (|\tau|) \land \\
\forall j \geq |\tau| - \text{taken} \land \text{cs} \gamma [\rho]_{\text{Suc} j} \}$$

An infinite execution $\sigma$ is said to be unfair to $\iota$ and indexed by $\tau$ if $\tau$ is a prefix of $\sigma$ and starting from the end of $\tau$, $\iota$ is not taken on $\sigma$ while being enabled infinitely often at the same time. The expression $\Gamma \land \text{CS} \land \gamma \land \tau$ represents the set of such infinite executions. Suppose $\sigma$ is in $\Gamma \land \text{CS} \land \gamma \land \tau$, if we count the number of times $\iota$ is enabled after $\tau$, this number will eventually transcend any natural number $i$. If $UF \land \text{cs} \land \gamma \land \tau \land i$ is the set of infinitely executions (the definition is given in Fig.6), in which the number of times $\iota$ is enabled after $\tau$ is no less than $i + 1$, then we have:

$$\Gamma \land \text{CS} \land \gamma \land \tau \subseteq UF \land \text{CS} \land \gamma \land \tau \land i$$

from this, we have $P(\Gamma \land \text{CS} \land \tau) \leq P(UF \land \text{CS} \land \tau \land i)$. If we can further prove for any natural number $i$ that:

$$P(UF \land \text{CS} \land \tau \land (i + 1)) < P(UF \land \text{CS} \land \tau \land i)$$  \hspace{1cm} (2)

then, from a standard result of mathematical analysis, we can prove $P(\Gamma \land \text{CS} \land \tau) = 0$.

Since all $UF$-sets are in the class $\text{sigma(Path,PA)}$, and therefore are not supported by finite sets of finite executions. This makes $UF$-sets difficult to deal with. The set function $BUF$ is defined in Fig. 6 to solve this problem, where the definition limits the value of $k$ to the parameter $up$, so that every $BUF \land \text{CS} \land \tau \land up \land i$ is supported by a corresponding finite set $SUF \land \text{CS} \land \tau \land up \land i$ which consists of finite executions. Therefore, we have

$$P(BUF \land \text{CS} \land \tau \land up \land i) = \sum_{\tau \in \text{SUF} \land \text{CS} \land \tau \land up \land i} \pi(\tau)$$  \hspace{1cm} (3)

Based on this and the investigating of the relationship between $SUF \land \text{CS} \land \tau \land up \land i$ and $SUF \land \text{CS} \land \tau \land up \land (i + 1)$ shown in Fig. 7, we managed to prove that:

$$P(BUF \land \text{CS} \land \tau \land up \land (i + 1)) \leq (1 - \text{bnd}) \times P(BUF \land \text{CS} \land \tau \land up \land i)$$ \hspace{1cm} (4)
\[
\text{Fig. 6. Definitions of sets of unfair executions}
\]

This together with the lemma

\[(\text{Aup. P (BUF} \iota \text{cs } \gamma) \implies \text{P (UF} \iota \text{cs } \gamma)\]

finally leads to the proof of

\[P (\text{UF} \iota \text{CS } \tau (i + 1)) \leq (1 - \text{bnd}) \ast P (\text{UF} \iota \text{CS } \tau i),\]

which entails the lemma in equation (2).

The detail of Fig. 7 is as follows: The dark gray box contains finite executions in \(\text{SUF} \iota \text{CS } \tau \text{ up } i\), while the light gray box contains those in \(\text{SUF} \iota \text{CS } \tau \text{ up } (i+1)\). Every finite execution is a branch of the execution tree. The height of the tree is limited by \(\text{up}\). Every finite execution in \(\text{SUF} \iota \text{CS } \tau \text{ up } (i+1)\) is extended from some \(\tau_1 \in \text{SUF} \iota \text{CS } \tau \text{ up } i\) by avoiding all the events to which \(i\) is assigned. Those events to which \(i\) is assigned are represented by dashed lines. Due to the avoidance of these dashed lines, it can be derived that:

\[
\left( \sum_{\tau_2 \in \text{SUF} \iota \text{CS } \tau \text{ up } (i+1)} \pi(\tau_2) \right) \leq (1 - \text{bnd}) \ast \pi(\tau_1)
\]

where \(\text{extending}(\tau_2, \tau_1)\) means \(\tau_2\) is an extension of \(\tau_1\), i.e., \(\tau_1\) is a suffix of \(\tau_2\). From this, we have:

\[(\sum \tau \in \text{SUF} \iota \text{CS } \tau \text{ up } (i+1). \pi(\tau)) \leq (1 - \text{bnd}) \ast (\sum \tau \in \text{SUF} \iota \text{CS } \tau \text{ up } i. \pi(\tau))\]
which, together with (3) can lead to (4).  
Packing all the above up, the final result in equation (1) is obtained.

Fig. 7. Relationship between $SUF \mathcal{C} S \uparrow \tau \uparrow i$ and $SUF \cup CS \uparrow \tau \uparrow (i+1)$

5 Related works

Paulson’s inductive approach for protocol verification[6] has been used to verify fairly complex security protocols [8, 7]. The success gives incentives to extend this approach to a general approach for concurrent system verifications. To achieve this goal, a method for the verification of liveness properties is needed. Paper [1] proposes such a method, where liveness proof rules are derived. The utility of this method has been confirmed by several applications[2–4]. This paper provides a probabilistic basis for the method. Parametric fairness is used in our method to simplify liveness proofs.

Probability space constructions for I/O automata can be found in[12, 13]. However, the execution sequence for I/O automata is an alternation of states and events, which is different from the pure event sequence used in our approach. This difference makes such works unsuitable for our purpose.

The probability space construction in this paper is heavily inspired by Hurd’s construction of probability space over sets of 0-1 sequences[11], but our construction is more general in the following senses:
- Our execution sequence contains events, the type of which could be very rich, while Hurd's sequence contains only 0 and 1.
- In our construction, the probability of the event to happen next depends on the current system state, while in Hurd’s construction, the probability of 0 and 1 is always 0.5.

To deal with these differences, we have to use different techniques.

As discussed in [14], there are many fairness notions with weak fairness and strong fairness as the standard ones. Nonstandard fairness notions, such as extreme fairness[5] and α-fairness[9], are introduced to reflect more adequately the underlying probabilistic execution, so that liveness proofs can be simplified. Baier proposed a general notion of fairness[10] which subsumes all existing fairness notions. This paper shows that parametric fairness is just another instance of this general fairness notion.

While standard fairness notions, such as weak fairness and strong fairness, can be expressed directly using LTL, extreme fairness and α-fairness and our parametric fairness cannot. Therefore, existing formulation of both extreme fairness and α-fairness are extra-LTL, at meta level, not mechanized. In this paper, LTL is embedded in Isabelle/HOL, therefore HOL serves as a mechanized meta language, in which nonstandard fairness notions can be expressed as well as its probabilistic model. We work under the belief that a mechanized meta theory may yield more reliability, maintainability and flexibility, as argued in Müller’s thesis[15].

Compared with [15], embeddings of temporal logic are very similar. However, our method uses a much simpler system model, and we believe this simpler model is adequate and more convenient to use in practice, as shown in the works[6,8,7,2-4]. Work [15] deals with standard fairness notions, while this paper deals with nonstandard fairness. Therefore, the liveness proof method is inherently different.

6 Conclusion

Inductive protocol verification is a method worth further extending. This paper together with [1-4] makes such an extension sound and practical. Works we are planning to do next include:

- Automate as much as possible the liveness proof procedure.
- Extract implementation code from the formal protocol specification.
- Investigate protocol refinement approaches.
- Do more case studies.
Acknowledgement: We have benefitted from the help of Joe Hurd and Stefan Richter.

References

Integrating Isabelle/HOL with Specware

Stephen J. Westfold
Kestrel Institute, Palo Alto, California, USA

Abstract. Isabelle/HOL is integrated with Specware in order to discharge proof obligations arising during Specware's specification and refinement process. Specware's proof obligations arise from use of predicate subtypes, termination conditions, and correctness of refinements as well as any explicit theorems. Specware specifications are structured into units called specs which correspond to Isabelle/HOL theories. Refinement is specified using spec morphisms and spec substitutions, a particular kind of colimit. We provide a system based on translating a Specware specification and its refinement into Isabelle/HOL theories, such that if Isabelle accepts all the translated theories, then the refinement is correct. Isabelle scripts for proving obligation theorems are embedded in Specware spec files so that proofs can be reused. The paper describes this translation and the issues that arise.

1 Introduction

Specware is a software specification and refinement system developed at Kestrel Institute based on higher-order logic and theory morphisms [1]. During Specware specification development and refinement to an implementation, many proof obligations are generated. Simple obligations can be discharged by Specware itself, but more complicated obligation theorems require an external theorem prover to discharge them. Previously, the main external theorem prover supported has been SNARK, an automatic first-order theorem prover [2]. This current work gives the user the alternative of proving Specware obligations using Isabelle/HOL, an interactive theorem prover for higher-order logic [3].

When SNARK succeeds in proving an obligation, everything is fine. However, often SNARK will fail to prove an obligation at first, either because the theorem is not true or the proof is too difficult for SNARK to find without further assistance. The first case, where the obligation theorem is not true, is common during the process of specification development, either because of incompleteness in the specification or because of mistakes made by the specifier. Finding the reasons for a failed proof can be an important part of the specification development process. Understanding proof failure has been difficult with SNARK, mainly because of the distance of the language of SNARK's (partial) proofs from that of Specware. This distance is because SNARK is a resolution-based prover using clausal form, whereas Specware uses higher-order logic with polymorphism.

Both Specware and Isabelle/HOL are based on higher-order logic with polymorphism, so Isabelle/HOL is a natural target for Specware. Specware declarations, definitions, axioms and theorems can be translated naturally to the cor-
responding Isabelle versions. The main additional feature of Specware's logic is predicate sub-typing where a type may be restricted by an arbitrary predicate.\footnote{PVS also has predicate subtyping, but lacks polymorphism [4].} Predicate subtyping can be used to restrict the domain of a function to legal inputs, and to give information about the the range of a function. These translate to Isabelle/HOL as extra axioms on definitions involving subtypes and theorems for applications of the function. The axioms typically state that for all inputs satisfying the predicate on the input type, the function applied to the inputs satisfies the predicate of the output type. The subtype theorems state that the arguments of the function satisfy the subtype predicate assuming information extracted from the context.

Specware and Isabelle/HOL share a similar import mechanism. Specware specifications are composed of specs which import other specs, whereas Isabelle/HOL has theories which import other theories. In addition Specware has the notion of spec morphisms\cite{5} which Isabelle/HOL does not, although there is recent work to add them in an extended version of Isabelle/HOL \cite{6}. A spec morphism is a mapping of op identifiers to op identifiers and type identifiers to type identifiers from a source spec to a target spec such that axioms in the source spec are theorems in the target spec. The target spec is a refinement of the source spec in that there is a target spec model for every source spec model, but not necessarily vice versa. Specware includes the colimit operation to provide a very general way of combining specs. We do not attempt to translate general colimits in this work, just spec substitutions which are a special case of colimit that cover most cases. A spec substitution takes a complex spec and a spec morphism where the source of the morphism is an import of the complex spec, and produces a copy of the complex spec with the source spec replaced by the target spec of the morphism. In this way, a spec substitution produces a refinement of a complex spec from a refinement of one of its components.

2 Embedding Proof Scripts in Specware

Simple extensions are made to the Specware syntax to allow Isabelle proof scripts to be embedded in Specware specs, and to allow the user to specify translation of Specware ops and types to existing Isabelle constants and types. The rest of the Specware system treats these as comments.

An embedded Isabelle proof script in a Specware spec consists of an introductory line beginning with \texttt{proof Isa}, the actual Isabelle script on subsequent lines terminated by the string \texttt{end-proof}. For example, the simple proof script \texttt{apply(auto) done} can be embedded as follows:

\begin{verbatim}
proof Isa
apply(auto)
end-proof
\end{verbatim}
If the last command before `end-proof` is not `done`, `qed` or `sorry`, the command `done` is inserted.

The proof script should occur immediately after the theorem or definition that it applies to. If the script applies to a proof obligation that is not explicit in the spec, then the name of the obligation should appear after `proof Isa`, on the same line.

If the user does not supply a proof script for a theorem then the translator will supply the script `apply(auto)` which is all that is required to prove many simple proof obligations.

3 Translation of Specware to Isabelle/HOL

Isabelle/HOL has types, constants and variables which correspond to Specware types, ops and variables.

3.1 Identifiers

All Specware variables are explicitly bound, whereas in Isabelle/HOL top-level variables are implicitly bound. This means that the translator needs to be concerned with renaming to avoid capture of implicitly bound variables such as `comp` and `o` by Isabelle/HOL constants of the same name.

Both systems have namespaces to avoid name confusion with a dot-notation for full names, but the namespace concepts are different, so the translator does not try to use the Isabelle dot-notation. Instead, an identifier such as `a.b` is translated to `a_b`. This scheme could possibly lead to name confusion if the user uses names containing double underbars.

Specware allows some non-alphabetic characters in the identifiers of ops and types that Isabelle/HOL does not, so these must be translated. For example, `equal?` is translated to `equal_p`.

3.2 Types

Most Specware types have directly corresponding Isabelle/HOL types:

- `Boolean → bool`
- `functions → functions`
- `products → pairs and tuples`
- `coproducts → constructor types (datatypes)`
- `type variables → type variables`
- `Char → char`
- `List → list`
- `String → string`

There is a slight mis-match with products and tuples in that in Specware products may be of arbitrary length whereas in Isabelle/HOL a tuple of more
than length 2 is really a structure of nested pairs. For construction this is transparent, but not for dereferences. For example, in Specware $x.2$ translates to $\text{snd } x$ if $x$ is a product of length 2, but it translates to $\text{fst(snd } x)$ if $x$ is a product of greater than length 2.

Coproducts are the same as constructor types except that a constructor with multiple arguments must be curried in Isabelle/HOL and uncurried in Specware. For example,

```plaintext
type bool =  
  | And (bool × bool)
  | Const Boolean
  | Neg bool
  | Var Nat
```

```plaintext
datatype bool =  
  And bool bool
  | Const bool
  | Neg bool
  | Var nat
```

Isabelle/HOL does not have predicate subtypes, so these are generally mapped to the supertype. The exception is that Specware types may be explicitly mapped to existing Isabelle/HOL types as described below, and this mapping allows a supertype to be mapped to a different type from the subtype with coercions functions to map between the two types. For example, in Specware, the type $\text{Nat}$ is a sub-type of $\text{Integer}$, but these are mapped to the separate Isabelle/HOL types $\text{nat}$ and $\text{integer}$. Mapping $\text{nat}$ to $\text{integer}$ would be correct but would make many proofs harder.

### 3.3 Terms

Isabelle/HOL and Specware share many of the same kinds of term, including case statements, lambda expressions, let expressions, if-then-else expressions, universal and existential quantifications, and function applications both in prefix form or infix. There are examples throughout the paper. In some cases the syntax is slightly different, but they are close enough that the meaning is obvious, so I will not describe the details. Specware has a letrec expression which is described below in the subsection on local definitions.

---

2 This example and some of the following are drawn from section 2.4.6 of the Isabelle/HOL tutorial [3].
3.4 Op Declarations and Definitions

A Specware definition may translate into one of three different kinds of Isabelle definitions: 
*defs*, *redefs* and *primrecs* (primitive recursions). Simple recursion on coproduct constructors translates to *primrec*, but if the function has multiple arguments, only if the function is curried. Other recursion translates to *redefs* which, in general, require a user-supplied measure function to prove termination. Non-recursive functions are translated to *defs*, except in some cases they are translated to *redefs* which allow more pattern matching.

For example, the Specware declaration and definition:

```latex
op bvalue:: boolex \rightarrow (Nat \rightarrow Boolean) \rightarrow Boolean
def bvalue be env =
case be of
| Const b \rightarrow b
| Var x \rightarrow env x
| Neg b \rightarrow \neg (bvalue b env)
| And(b,c) \rightarrow bvalue b env \land bvalue c env
```

```latex
consts bvalue :: “boolex \Rightarrow (nat \Rightarrow bool) \Rightarrow bool”
primrec
“bvalue (Const b) env = b”
“bvalue (Var x) env = env x”
“bvalue (Neg b) env = (\neg (bvalue b env))”
“bvalue (And b c) env = (bvalue b env \land bvalue c env)”
```

Recursive functions that are translated to *redefs* can have a measure function specified on the *proof Isa* line, by including it between double-quotes. For example:

```latex
proof Isa “measure (\lambda (wrd,sym). length wrd)” end–proof
```

There are examples of different kinds of definition below.

3.5 Axioms and Theorems

Specware axioms and theorems are translated naturally to Isabelle/HOL axioms and theorems. For example,

```latex
theorem valif is
\forall (b,env) valif (bool2if b) env = bvalue b env
proof Isa
apply(induct_tac b)
apply(auto)
```
end-proof

is translated to

\textbf{theorem} \textit{valif}:

\textit{"valif (bool2if b) env = bvalue b env"}
apply\(\textit{induct_tac b}\)
apply\(\textit{auto}\)
done

Annotations for theorems may be included on the \textit{proof Isa} line. For example,

\textbf{theorem} \textit{Simplify_valif_normif} is

\textit{\(\forall (b, env, t, e) \ valif \ (\text{normif} \ b \ t \ e) \ env = \ valif \ (IF \ b \ t \ e) \ env\) }
\textbf{proof \ Isa [simp]}
apply\(\textit{induct_tac b}\)
apply\(\textit{auto}\)
done

translates to

\textbf{theorem} \textit{Simplify_valif_normif [simp]}:

\textit{\"valif (normif b t e) env = valif (IF b t e) env\"}
apply\(\textit{induct_tac b}\)
apply\(\textit{auto}\)
done

In this example we see that universal quantification in Specware becomes, by default, implicit quantification in Isabelle. This is normally what the user wants, but not always. The user may specify the variables that should be explicitly quantified by adding a clause like \(\forall e\). to the \textit{proof Isa} line. For example,

\textbf{theorem} \textit{Simplify_valif_normif} is

\textit{\(\forall (b, env, t, e) \ valif \ (\text{normif} \ b \ t \ e) \ env = \ valif \ (IF \ b \ t \ e) \ env\) }
\textbf{proof \ Isa [simp] \(\forall t\).}
apply\(\textit{induct_tac b}\)
apply\(\textit{auto}\)
done

which translates to

\textbf{theorem} \textit{Simplify_valif_normif [simp]}:

\textit{\"\(\forall t::ifex\) (e::ifex). valif (normif b t e) env = valif (IF b t e) env\"}
apply(induct_tac b)  
apply(auto)  
done

It is sometimes necessary to add explicit type annotations to allow the Isabelle type-checker to resolve overloading.

### 3.6 Subtype Axioms and Obligations

If the result of an op is a subtype then an axiom is produced to that effect. For example:

```plaintext
op n: \{i: Nat \mid i > 0\}  
op f(x: Nat): \{i: Nat \mid i > 0\}
```

translates to

```plaintext
consts n :: "nat"  
axioms n_subtype_constr:  
  "n > 0"
consts f :: "nat ⇒ nat"  
axioms f_subtype_constr:  
  "f i > 0"
```

For references to functions with take a subtype as an argument, a subtype obligation is generated for each application of the function to ensure that the subtype predicate holds.

For example, the declaration of the \textit{nth} function is as follows:

```plaintext
op nth: \[a\] \{(l,i) : LIST \times Nat \mid i < \text{n} \} \rightarrow a
```

Note that, to express this subtype restriction the \textit{nth} cannot be a curried function because Specware does not have dependent types. However, we do translate \textit{nth} to the corresponding Isabelle function \textit{!} which is curried.

For example, from

```plaintext
op L: List Nat = [1,2,3]  
proof Isa [simp] end-proof  
op L2: Nat = nth(L,2)
```

we get

```plaintext
consts L :: "nat list"  
defs L_def [simp]: "L ≡ [1,2,3]"
```
consts \( L2 :: \text{"nat"} \)

theorem \( L2\_Obligation\_subsort: \)
\[ 2 < \text{length } L \]
apply(auto)
done
defs \( L2\_def: \text{"} L2 \equiv L \downarrow 2 \text{"} \)

3.7 Local Recursive Functions

Local recursive functions are allowed in Specware but not in Isabelle/HOL. Therefore these functions are lifted to the top level using the technique of lambda-lifting \([7]\) which converts free variables to extra parameters.

For example, the \texttt{tabulate} function:

\begin{verbatim}
  op [a] tabulate(n: Nat, f: Nat \rightarrow a): List a =
  let def tabulateAux (i : Nat, l : List a) : List a =
    if i = 0 then l
    else tabulateAux(i-1,Cons(f(i-1),l)) in
  tabulateAux(n,[])

  proof Isa tabulate \texttt{tabulateAux} "measure (\lambda(i,l,f) \cdot i)" end-proof
\end{verbatim}

is translated to

\begin{verbatim}
  recdef \texttt{tabulate} \texttt{tabulateAux} "measure (\lambda(i,l,f) \cdot i)"
  "tabulate\_tabulateAux(0,l,f) = l"
  "tabulate\_tabulateAux(Suc i,l,f)
    = tabulate\_tabulateAux(i,Cons(f i) l,f)"

  const tabulate :: "nat \times (nat \Rightarrow 'a) \Rightarrow 'a list"

  recdef \texttt{tabulate} "{}"
  "tabulate(n,f) = tabulate\_tabulateAux(n,[]).f"
\end{verbatim}

Note the translation of the \texttt{if \ i = 0 \ then \ ... \ else \ ...} into cases on \texttt{0} and \texttt{Suc i}. Specware does not allow a case split on successor because naturals are defined as a subtype of integers and not constructed from zero and successor.

3.8 Logic Morphisms

We wish to exploit existing Isabelle/HOL libraries, so we provide a mechanism for mapping Specware op and type identifiers to Isabelle/HOL constant and type identifiers. Any definitions in Specware for these ops must be theorems in the Isabelle/HOL theory for this mapping to be correct.

A translation table for Specware types and ops is introduced by a line beginning \texttt{proof Isa Thy Morphism} followed optionally by an Isabelle theory
(which will be imported into the translated spec), and terminated by the string \textbf{end-proof}. Each line gives the translation of a type or op. For example, for the Specware Option theory we have:

\begin{verbatim}
proof Isa Thy_Morphism
  type Option.Option → option
  Option.mapOption → option_map
end-proof
\end{verbatim}

A type translation begins with the word \texttt{type} followed by the fully-qualified Specware identifier, \texttt{→} and the Isabelle identifier. If the Specware type is a sub-type, you can specify coercion functions to and from the super-type in parentheses separated by commas. Note that by default, sub-types are represented by their super-type, so you would only specify a translation if you wanted them to be different, in which case coercion functions are necessary. Following the coercions functions can appear a list of overloaded functions within square brackets. These are used to minimize coercions back and forth between the two types.

An op translation begins with the fully-qualified Specware identifier, followed by \texttt{→} and the Isabelle constant identifier. If the Isabelle constant is an infix operator, then it should be followed by \texttt{Left} or \texttt{Right} depending on whether it is left or right associative and a precedence number. Note that the precedence number is relative to Specware’s precedence ranking, not Isabelle’s. Also, an uncurried Specware op can be mapped to a curried Isabelle constant by putting \texttt{curried} after the Isabelle identifier, and a binary op can be mapped with the arguments reversed by appending \texttt{reversed} to the line.

For Specware’s Integer spec we have the logic morphism

\begin{verbatim}
proof Isa Thy_Morphism Presburger
  type Integer.Integer → int
  type Nat.Nat → nat (int,nat) [+,*\texttt{div,rem},\texttt{\le,\ge,\lt,\gt,abs,min,max}]
  Integer.+ → + Left 25
  Integer.– → – Left 25
  IntegerAux.– → –
  Integer.\times → \times Left 27
  Integer.\texttt{div} → \texttt{div} Left 26
  Integer.\texttt{rem} → \texttt{mod} Left 26
  Integer.\texttt{\le} → \texttt{\lt} \texttt{\le} Left 20
  Integer.\texttt{\lt} → < Left 20
  Integer.\texttt{\ge} → \texttt{\gt} \texttt{\ge} Left 20
  Integer.\texttt{\gt} → > Left 20
  Integer.\texttt{min} → \texttt{min curried}
  Integer.\texttt{max} → \texttt{max curried}
end-proof
\end{verbatim}

Note that the list of overloaded functions does not include \texttt{–} because Is-
abelle’s definition of \(\cdot\) on naturals is not the same as Specware’s if the second argument is larger than the first: for Isabelle the value is 0 whereas for Specware it is a negative integer. However, if the consumer of the subtraction is expecting a natural then we know there is a proof obligation that ensures this, so in this case we do translate Specware’s \(\cdot\) to Isabelle’s.

### 3.9 Spec Morphisms and Spec Substitutions

The obligations theory in Isabelle/HOL of a spec morphism is simply a theory that imports the translation of target spec of the morphism and includes the axioms of the source spec translated along the morphism as theorems to be proven. Named proof scripts for the theorems follow the morphism.

For example, the morphism \(AB.M\) from spec \(A\) to spec \(B\) in the following:

\[
A = \text{spec} \\
\quad \text{op } f : \text{Nat }\rightarrow\text{Nat} \\
\quad \text{axiom } f_{\text{pos}} \text{ is } \forall (x) f \ x > 0 \\
\quad \text{endspec}
\]

\[
B = \text{spec} \\
\quad \text{op } g(i : \text{Nat}) : \text{Nat} = i + 2 \\
\quad \text{endspec}
\]

\[
AB.M = \text{morphism } A \rightarrow B \{f \mapsto g\}
\]

\[
\text{proof } \text{ Isa } f_{\text{pos}} \\
\quad \text{apply } (\text{auto simp add: } g_{\text{def}}) \\
\text{end } - \text{proof}
\]

has the obligation theory:

\[
\text{theory } AB.M \\
\text{imports } B \\
\text{begin} \\
\text{theorem } f_{\text{pos}}: \\
\quad "g \ j > 0" \\
\quad \text{apply } (\text{auto simp add: } g_{\text{def}}) \\
\quad \text{done} \\
\text{end}
\]

The syntax of a spec substitution is a spec term followed by a morphism in square brackets. For example, the spec \(D\) is defined as the substitution of morphism \(AB.M\) applied to the spec \(C\) in the continuation of the current example:

\[
C = \text{spec} \\
\quad \text{import } A
\]
\texttt{op f2(i: Nat): Nat = f(f i)}

\texttt{endspec}

\[ D = \text{C[AB}_M\text{]} \]

The translation of spec \( D \) is just spec \( C \) with \( A \) replaced by \( B \) and performing the renaming of the morphism \( AB_M \) giving the theory:

\begin{verbatim}
theory D
  imports B
begin
  consts f2 :: "nat ⇒ nat"
 defs f2_def: "f2 i ≡ g (g i)"
end
\end{verbatim}

Note that if the spec to be replaced is deeply embedded in the import structure, then it is necessary to make copies of all the imported specs that import this spec, and translate them to Isabelle/HOL theories. For a large specification and refinement there may be many substitutions applied in sequence to the top level spec, implicitly using many intermediate specs. It is important to generate the specs required by the final refinement and their corresponding Isabelle/HOL theories without generating all the intermediates.

4 Current Restrictions and Future Work

This initial translator has a number of limitations. It should translate all Specware specs but not all translated definitions and constructs will be accepted by Isabelle/HOL. In particular, only case expressions that involve a single level of pattern-matching on constructors are accepted. An exception, is that some nesting is allowed in top-level case expressions that are converted into definition cases. Mutual recursion is not currently supported. The translator currently targets the 2006 release version of Isabelle. The next version of Isabelle includes a new function package that should allow more Specware definitions to be translated. Also, it allows termination to be proved for a subdomain which is a natural match for a Specware function whose domain is a predicate subtype.

Bortin, Johnsen and Lüth have developed an extension of Isabelle that includes theory morphisms [6]. This would provide a natural target for a translation of Specware’s spec morphisms. Morphism obligations are proved, but there is a meta-theorem about morphisms that all theorems in the source spec of the morphism, are theorems in the target spec when translated by the renaming of the morphism. We do not currently exploit this powerful property, unlike this Isabelle extension which realizes these theorems by translating proofs along the morphism. By targeting this extension we would gain this property. A concern is that the price of this extension is that low-level proofs must be stored, which
could be expensive, especially for a large specification with many spec substitutions.

In the future we wish to allow results from Isabelle inference to be returned to Specware. Witness-finding can be used in a number of ways during algorithm synthesis, for example, to instantiate a function in a program schema [8]. If we find a witness for an existential during a proof in Isabelle, we need to translate it back into Specware term, so it can be used to give a definition in a refined spec. This back-translation is straightforward, as the term language for Specware and Isabelle are very similar. The only significant issue is the different name-space rules of the two systems. In particular, translation of Specware’s qualified names need to be invertible.

5 Conclusions

Initial experience with using Isabelle/HOL to discharge proof obligations has been positive. The Isabelle/HOL translation of a Specware spec is very readable with direct correspondence between many elements. The translation includes extra theorems, mainly for sub-type obligations, but their names and location make them easy to connect to their origin. The main concern with using Isabelle/HOL compared to using SNARK was that the user would have to be concerned with giving proofs for many simple obligations. In our limited experience, the auto tactic is able to discharge most of the obligations that SNARK was able to discharge automatically. By making this the default tactic, we have avoided cluttering Specware specs with trivial proofs. In addition, the means of controlling the proof process in Isabelle/HOL are more intuitive than with SNARK, and the reasons for failure of a proof are more apparent, and it is easier to control a more complicated proof. Termination proofs, in particular, have been much easier in Isabelle/HOL, with the user just providing a measure function.

We have used the Specware to Isabelle/HOL integration with tutorial examples and Specware’s libraries. This has revealed several subtle bugs in the specifications. We plan to use this system to prove the correctness of more significant refinements, and to help in automatically generating correct refinements.

References


Author Index

Abed, Sa’ed ...................... 1
Ait Mohamed, Otmane .......... 1
Al Sammane, Ghiath .......... 1
Amjad, Hasan ................. 18
Benzmueller, Christoph ....... 33
Brucker, Achim D. ............ 49
Corbineau, Pierre ............. 65
Dawson, Jeremy .............. 81
Gast, Holger .................. 97
Gordon, Michael J. C. ......... 170
Gottliebsen, Hanne .......... 113
Haftmann, Florian ........... 128
Kammueller, Florian .......... 144
Le Roux, Stephane .......... 154
Lueth, Christoph .......... 178
Maeder, Christian .......... 178
Mossakowski, Till .......... 178
Myreen, Magnus .......... 170
Nipkow, Tobias ........... 128
Torrini, Paolo .............. 178
Wang, Jinshuang .......... 194
Westfold, Stephen .......... 210
Wolff, Burkhart ............ 49
Yang, Huabing ........... 194
Zhang, Xingyuan .......... 194
Zhang, Yusen .......... 194